# Transaction languages for database update and specification

Serge Abiteboul, Victor Vianu

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 715

# TRANSACTION LANGUAGES FOR DATABASE UPDATE AND SPECIFICATION

**Serge ABITEBOUL**
**Victor VIANU**

**SEPTEMBRE 1987**

# TRANSACTION LANGUAGES FOR DATABASE UPDATE AND SPECIFICATION

Langages transactionnels pour
mises-à-jour et spécification de bases de données

Serge Abiteboul
I.N.R.I.A.
Domaine de Voluceau-Rocquencourt
78153 Le Chesnay
FRANCE
(serge@inria.inria.fr)

Victor Vianu[1]
CSE Department,
University of California at San Diego
La Jolla, CA 92093
USA
(vianu@sdcsvax.ucsd.edu)

August 15, 1987

*Abstract:* Relational database updates, and specification using updates are studied. Completeness criteria for transaction languages are introduced, based on their updating and specification capability. Then, several specific transaction languages are defined: TL (non-deterministic), and detTL (deterministic). Restrictions of these languages are also considered. The completeness of TL and detTL w.r.t. several proposed completeness criteria is shown. The power of the restricted languages is characterized. Some connections with existing notions of query completeness, and corresponding query languages, are exhibited.

*Résumé:* Cette étude porte sur les mises-à-jour dans les bases de données relationnelles, et la spécification utilisant ces mises-à-jour. Des critères de complétude pour languages transactionnels sont introduits basés sur leurs puissances de mise-à-jour et de spécification. Plusieurs languages de transactions sont définis: TL (non-déterministe) et detTL (déterministe). Des restrictions de ces languages sont aussi considérées. La complétude de TL et detTL en rapport avec plusieurs notions de complétudes est démontrée. Le puissance des restrictions des languages est caractérisée. De plus, des liens avec des notions de complétude de requêtes, et certains languages de requêtes sont mis en évidence.

[1] This work was performed in part while the author was visiting at I.N.R.I.A. The author was supported in part by the National Science Foundation under grant number IST-8511538.

## INTRODUCTION.

While query languages have been extensively studied, update languages have only recently become the object of formal investigation. Recent research has focused on the use of transactions as specification tools [AV2,AV3,Br,CCF,SS] and on the computational power of transactions [AV2,AV4]. Results in [AV2] suggest notions of completeness for transaction languages, which are analogs of the notions of completeness for query languages but are particular to transactions. In this paper, we consider several such notions, and present simple and natural transaction languages which are complete with respect to these criteria.

Several notions of completeness are proposed, which correspond to the use of transactions as updating tools, and as specification tools:

- The notion of "update completeness", concerns the update capability of a given transaction language, and is closely related to the "query completeness" of Chandra and Harel [CH1]. However, the notions of update and query completeness are incomparable. We argue that our notion of completeness is more natural in a database context.
- The notion of "specification completeness", is related to the concept of transactional schema, introduced in [AV2]. In a transactional schema, the set of valid updates is specified using a set of admissible transactions. Specification completeness concerns the ability of transactional schemas to describe any "reasonable" static or dynamic database semantics.

The completeness criteria involves computability, and some isomorphism properties considered in [B,P,HY,AU,CH1]. Two versions of update and specification completeness are defined, depending upon the determinism or non-determinism of the updates that are considered.

The transaction languages that we propose use the elementary operations of tuple insertion and deletion, and a "while" construct. Finally, another construct ("with new") allows the random assignment to a domain variable of a new domain value, not in the original database. The use of arbitrary, "invented" values raises the issue of "safety" of transactions, similar to safety of queries [U,M]. We show that unsafe transactions are necessary in order to achieve the desired computational power. We compare our approach to that of Chandra and Harel, who encode information in the number of attributes of relation schemas.

We obtain results on safety which are comparable to results in [U,M] for safe tuple calculus expressions. In particular, we exhibit a syntactic restriction which guarantees safety, and show that for each safe transaction, there exists an equivalent "syntactically safe" transaction. Other aspects of the constructs are also examined, such as a trade-off between the use of negative predicates in conditions, and explicit deletion.

With the constructs described above, we obtain a non-deterministic Transaction Language (TL). A deterministic counterpart (detTL) is obtained be providing deterministic semantics to the while construct, and by disallowing the use of randomly invented values in the result.

The main results of the paper concern the non-deterministic and deterministic update and specification completeness of TL and detTL, respectively.

We define restrictions of TL and detTL based on the safe use of invented values, and characterize their computational power. First, we consider the language obtained by disallowing (syntactically) the presence of invented values in the result of a TL transaction. We show that this language yields a natural subclass of the non-deterministic updates, i.e., the class of updates where the non-determinism is finite in a certain sense. Also, by totally disallowing the use of invented values in TL, we obtain a language which computes exactly the NPSPACE updates. An analogous restriction on detTL yields a language which encompasses all fixpoint queries and has exactly the power of the language *LE* [CH1]. As a consequence, the language thereby obtained expresses strictly less than the PSPACE updates. (We show that exactly the PSPACE updates are obtained, under the assumption that an ordering of the active domain is provided in the database.)

The paper is organized as follows. In the first section, the notions of database update and specification are introduced. Specification using transactions is compared to specification using traditional static constraints. Update and specification completeness criteria for transaction languages are exhibited. The transaction languages are presented in Section II. These languages are: TL (non-deterministic), safe restrictions of TL (WTL and STL), detTL (deterministic) and a safe restriction of detTL (SdetTL). The power of these languages with respect to updates is considered in Section III, and their specification power in Section IV.

# 1. UPDATE AND SPECIFICATION.

We start by reviewing some basic database terminology and notation [U]. We introduce the fundamental notions of database updates, transition sets, and families. We present completeness criteria for transaction languages with respect to updating and specification capability.

## 1.1 Preliminaries

We assume the existence of three infinite and pairwise disjoint sets of symbols: the set **att** of *attributes*, the set **dom** of *constants*, and the set **var** of *variables*. A *relational schema* is a finite set of attributes. A *tuple* over a relational schema R is a mapping from R into **dom** $\cup$ **var**. A *constant tuple* over a relational schema R is a mapping from R into **dom**. An *instance* over a schema R is a finite set of constant tuples over R. A *database schema* is a finite set of relational schemas. An *instance* **I** over a database schema **R** is a mapping from **R** such that for each R in **R**, **I**(R) is an instance over R. In general, we use A, B, C,... for attributes, a, b, c,... for constants, x, y, z,... for variables, and r, s, t,... for tuples. We usually denote relational schemas by R, S,..., database schemas by **R**, **S**,..., and database instances by **I**, **J**,.... The set of all instances over a schema **R** is denoted by **inst**(**R**).

For each set X (relation, schema, database schema,...), the cardinality of X is denoted by #(X). For each instance **I** over **R**, #(**I**) denotes $\sum_{R \in \mathbf{R}} \#(\mathbf{I}(R))$. For each instance **I**, the set of constants occurring in **I** is denoted by **const**(**I**).

Let **R** be a database schema.

A *functional dependency* (FD) [C] over **R** is an expression $X \rightarrow Y$ where X and Y are subsets of a relational schema R in **R**. An instance **I** of **R** satisfies $X \rightarrow Y$ iff for each R in **R** containing $(X \cup Y)$, and for each tuples u, v in **I**(R), u[X]=v[X] implies u[Y]=v[Y].

An *inclusion dependency* (IncD) [CFP] over **R** is an expression $R(A_1...A_m) \subseteq S(B_1...B_m)$ where R, S are in **R**, $m \geq 1$, each $A_i$ is in R, and each $B_i$ is in S. (We also assume that $A_i \neq A_j$ and $B_i \neq B_j$ if $i \neq j$.) An instance **I** of **R** satisfies $R(A_1...A_m) \subseteq S(B_1...B_m)$ iff for each u in **I**(R) there exists v in **I**(S) such that $u(A_i) = v(B_i)$ for each i.

As in [BV], a *total tuple generating dependency* (total TGD) or an *equality generating dependency* (EGD) g is a sentence in relational calculus of the form

$$\forall\ x_1,...,x_p\ (R(u_1)/\backslash.../\backslash R(u_k) \Longrightarrow \xi)$$

where $R \in \mathbf{R}$, $u_i$ $(1 \leq i \leq k)$ are tuples containing constants and variables, and all variables are universally quantified. For EGD's, $\xi$ is of the form x=y, where x and y are variables occurring in some $u_i$, $u_j$, or x is a variable and y is a constant. For total TGD's, $\xi$ is of the form R(u), where u is a tuple of variables occurring in the $u_i$, and constants. (Note that an FD is a particular case of EGD, and a join dependency is a total TGD.) Satisfaction of total TGD's and EGD's is defined in the obvious manner.

In this section, we do not consider any particular database language. We introduce general database concepts in the context of the relational model. These notions can be extended to other models as well.

### 1.2 Updates

The first concept that we consider is the notion of "update". We view updates as transformations of database states into other database states. Such transformations include updates in the usual sense (i.e., modifications of the database state), but also queries or restructurings of the database. We take the view that the difference between updates and queries concerns primarily the interpretation of the results, rather than the computation involved. Therefore, we do not emphasize a distinction between queries and updates. However, the word "update" is chosen here since our transaction languages use tuple insertions and deletions, which are traditionally viewed as update operations.

In general, transformations are not required to be mappings. While in some cases, the transformations considered will be mappings, in others, non-determinism is allowed, and the transformation is described as a relation between database states. In a database context, such transformations are not arbitrary. Clearly, the relation between states has to be at least recursively enumerable (r.e.). It is usually required that instances over a *fixed* schema be related to instances over another fixed schema. Finally, it is also required that constants (except perhaps for a fixed number) be essentially viewed as uninterpreted. The last property has been independently introduced by [B,P,HY,AU,CH1] under different names and with minor differences. We use here the terminology of [HY].

*Definition:* Let **R** and **S** be database schemas, and C a finite set of constants.

- A subset $\tau$ of $\mathbf{inst(R)} \times \mathbf{inst(S)}$ is *C-generic* iff for each bijection $\rho$ over **dom** which is the identify on C, $(\mathbf{I},\mathbf{J}) \in \tau$ iff $(\rho(\mathbf{I}),\rho(\mathbf{J})) \in \tau$.
- A mapping $\tau$ from $\mathbf{inst(R)}$ to $\mathbf{inst(S)}$ is *C-generic* iff its graph is C-generic.
- A set X of instances is C-generic iff for each bijection $\rho$ over **dom** which is the identity on C, $\mathbf{I} \in \mathbf{X}$ implies $\rho(\mathbf{I}) \in \mathbf{X}$.

Mappings considered in databases are in general C-generic. For instance, relational calculus queries (selection using $=$, $\neq$ only) and fixpoint queries are C-generic for some C. Insertions, deletions and modifications are also C-generic for some C.

We now define the class of non-deterministic updates:

*Definition:* Let **R** and **S** be database schemas. A *(non-deterministic) update* (from **R** to **S**) is a subset of $\mathbf{inst(R)} \times \mathbf{inst(S)}$ which is r.e., and C-generic for some finite C.

Note that, although not allowed by traditional database systems, non-deterministic updates (or queries) arise quite naturally. For instance, consider the query: "Find one café at the intersection of Blvd. St. Michel and Blvd. St. Germain". Clearly, the query may have different outcomes since it is well known that there are several cafés at that intersection. Similarly, the update "Assign as an undergraduate advisor some junior professor who was not advisor in the past three years", is also non-deterministic.

In general, a given state can be transformed into infinitely many possible states by a non-deterministic update. We will see that the class of non-deterministic updates such that for each state, the number of possible next states is finite, arises naturally. This suggests the following definition:

*Definition:* Let **R** and **S** be database schemas. A *finitely non-deterministic update* $\tau$ (from **R** to **S**) is a non-deterministic udpate such that

$$\forall\ \mathbf{I}, \exists\ \mathrm{n}, \#\{\mathbf{J} \mid (\mathbf{I},\mathbf{J}) \in \tau\} \leq \mathrm{n}.$$

Note that, due to the C-genericity of updates, $\tau$ is finitely non-deterministic iff

$$(\mathbf{I},\mathbf{J}) \in \tau \Longrightarrow \mathbf{const(J)} \subseteq \mathbf{const(I)} \cup \mathrm{C}.$$

Of course, deterministic updates are of particular importance. They are defined next.

*Definition:* Let **R** and **S** be two database schemas. A *deterministic update* (from **R** to **S**) is a mapping from **inst(R)** into **inst(S)** which is partial recursive, and C-generic for some finite C.

We now illustrate the notions defined above:

*Example 1.2.1:* Let R be a unary relation. Consider the following operations.

$t_1$ = insert(5),

$t_2$ = delete_random_tuple,

$t_3$ = insert_random_tuple.

The first update is deterministic. The second one is finitely non-deterministic but not deterministic. The last one is not finitely non-deterministic.

With the three classes of updates defined above, we can now introduce completeness criteria for transaction languages based on their capability to express transformations in these classes. In our discussion of completeness, we will use the term "transaction language" without defining it formally. Intuitively, a *transaction language* L specifies a set of transactions. For each transaction t in L, its semantics is given by an update called the *effect* of the transaction, and denoted by $\text{eff}_L(t)$. Note that $\text{eff}_L(t)$ is a mapping if the transaction is deterministic, and a relation otherwise. Now we have:

*Definition:* Let L be a transaction language. Then

- L is *(non-deterministic) update complete* if
$$\{\text{eff}_L(t) \mid t \text{ in } L\} = \text{set of non-deterministic updates;}$$
- L is *finitely non-deterministic update complete* if
$$\{\text{eff}_L(t) \mid t \text{ in } L\} = \text{set of finitely non-deterministic updates; and}$$
- L is *deterministic update complete* if
$$\{\text{eff}_L(t) \mid t \text{ in } L\} = \text{set of deterministic updates.}$$

In Section 3, we exhibit languages (TL, WTL, detTL) which are shown to be non-deterministic, finitely non-deterministic, and deterministic update complete, respectively.

## 1.3 Dynamic specification

Database specification using transactions has been studied in [AV2,AV3]. Intuitively, a specification describes the valid database states (static specification), and valid database evolution (dynamic specification). More precisely, a dynamic specification defines the valid state transitions for a given database. A dynamic specification is meant to reflect semantics involving change in the application environment. In some sense, a dynamic specification is more powerful than a static specification. Indeed, each dynamic specification induces a static specification: given a dynamic specification, the valid instances are those which can be reached from some start instance. The converse is not true. In this section, we consider dynamic specification. We formalize this notion using the concept of transition set (set of valid state transitions).

*Definition:* Let **R** be a database schema. A *transition set* is a binary relation over inst(**R**) which is r.e. and C-generic for some finite C.

A *(dynamic) specification language* L consists of words (dynamic specifications), each of which is used to specify a set of valid transitions. Thus, for each T in L, its semantics is given by a transition set denoted by $tran_L(T)$. In this paper, we consider specifications languages closely related to our transaction languages. More precisely, the specification languages that we consider provide a set of transaction "procedures"; valid transitions are then defined as effects of calls to the given procedures. This is illustrated in the following example.

*Example 1.3.1:* Let R = {A,B} be a database schema. Suppose that the legal operations are "calls" to the procedure:

```
procedure Insert(x,y)
begin
for each (x,z) in R, delete(x,z);
insert(x,y)
end
```

Let L be the underlying language with the obvious semantics. Then, $tran_L$(Insert) is the set of transitions (**I**, **J**) such that **J** is obtained from **I** by executing the procedure Insert on **I** with some parameters a and b. For instance, ( { <1,2>, <2,5>, <3,4> } , { <1,2>, <2,8>, <3,4> } ) is in the transition set $tran_L$(Insert). (The call used here is Insert(2,8).)

In the spirit of Example 1.3.1, a transaction language (TL) is later used to obtain a specification language based on transaction procedures.

We now define a completeness criteria for specification languages:

*Definition:* Let **R** be a database schema, and L a specification language. Then L is *(dynamic) specification complete* if

$$\{\text{tran}(T) \mid T \in L\} = \text{set of transition sets.}$$

In Section 4, we prove that the specification language obtained from TL (based on procedure calls) is specification complete. Clearly, other kinds of specification languages related to transaction languages can be considered. For instance, one can associate with transactions triggering mechanisms responsible for maintaining the integrity of the database. More declarative specification languages (e.g., based on temporal logic or dynamic constraints) can also be considered. In the present paper, we will mainly be interested in specification languages obtained by considering transaction procedures. However, the concepts and results in this section apply to any specification language.

We now present a special class of transition sets, called "bounded". Let $\tau$ be an arbitrary transition set. Suppose that $(\mathbf{I},\mathbf{J})$ is in $\tau$. In general, there is no bound on the number of constants that may occur in **J** and not in **I**. However, it is often reasonable to require that the constants newly introduced in a state transition be explicitly specified in the transaction itself, or by the user (say as parameters). In particular, this "safety" condition is certainly required if the transactions are deterministic and C-generic. In such a case, the constants that occur in **J** and not in **I** must be in the finite set C. Therefore, the number of constants appearing in **J** and not in **I** is bounded by the size of C. This leads to the following definition:

*Definition:* A transition set $\tau$ is *bounded* if

$$\exists\ n,\ \forall\ (\mathbf{I},\mathbf{J}) \in \tau,\ \#(\mathbf{const(J)} - \mathbf{const(I)}) \leq n.$$

A language L is *bounded specification complete* if

$$\{\text{tran}(T) \mid T \in L\} = \text{set of bounded transition sets.}$$

In Section 4, we provide a language (WTL) which is bounded specification complete.

The last notion of specification completeness considered in the paper (and the more restrictive) is that of deterministic specification completeness. As indicated by its name, this notion corresponds to specifications based on deterministic programs. Although the definition may seem mysterious at first glance, the imposed restriction should become clear in Section 4.

*Definition:* A transition set H is *deterministic* if for some $n \geq 0$, and some finite set C of constants,

$$H = \cup \{f(\tau_i) \mid 1 \leq i \leq n, f \text{ is an isomorphism, } f|_C = id\},$$

where each $\tau_i$ is a deterministic update which is $(C_i \cup C)$-generic for some finite $C_i$ $(1 \leq i \leq n)$.

Intuitively, the deterministic updates $\tau_i$ correspond to n deterministic procedures. The set C corresponds to the set of constants in the procedures, and the $C_i$ to the parameters. The bijection f corresponds to different calls to the deterministic procedures, which affect the constants in the $C_i$ but not in C.

*Definition:* A specification language L is *deterministic specification complete* if

$$\{ \text{tran}_L(T) \mid T \text{ in } L \} = \{ \text{deterministic transition sets} \}$$

In Section 4, we exhibit a language (detTL) which is deterministic specification complete.

The following simple result relates the various notions of transition sets and thus of dynamic specification completeness.

*Proposition 1.3.1:*

(1) any deterministic transition set is bounded;

(2) there are transition sets which are not bounded;

(3) there are transition sets which are bounded and not deterministic;

*Proof:* (1) is obvious.

To see (2), consider a schema **R**, and the transition set over **R**:

$$\tau = \{ (\mathbf{I},\mathbf{J}) \mid \#\text{const}(\mathbf{J}) = 2 \cdot \#\text{const}(\mathbf{I}) \}.$$

It can be shown that $\tau$ is a transition set and is not bounded.

To see (3), consider the transition set over some relational schema R defined by:

$$\tau = \{ (\mathbf{I},\mathbf{J}) \mid \mathbf{J} \subseteq \mathbf{I} \}.$$

Then $\tau$ is bounded but not deterministic. □

The previous proposition will allow the comparison of specification powers of various languages presented in the paper.

To conclude this section, we exhibit a connection between specification and update completeness. Note that a transaction language can also be viewed trivially as a dynamic specification language. Indeed, for each transaction language L, consider the specification language

$$\Lambda(L) = \{t \in L \mid \text{eff}_L(t) \subseteq \text{inst}(\mathbf{R}) \times \text{inst}(\mathbf{R}) \text{ for some } \mathbf{R}\},$$

with semantics given by

$$\text{tran}_{\Lambda(L)}(t) = \text{eff}_L(t)$$

for each t in L. There is clearly a connection between the update completeness of L, and the specification completeness of $\Lambda(L)$:

*Proposition 1.3.2:* Let L be a transaction language. (1) If L is update complete, then $\Lambda(L)$ is dynamic specification complete. (2) The converse does not hold.

*Proof:* (1) Obvious. Consider (2). Let L be an update complete transaction language. Let L' = $\{t \mid \text{for some } \mathbf{R}, \text{eff}_L(t) \subseteq \text{inst}(\mathbf{R}) \times \text{inst}(\mathbf{R})\}$ be the transaction language such that $\text{eff}_{L'}(t) = \text{eff}_L(t)$ for each t in L'. Then L' is not update complete, and $\Lambda(L')$ is dynamic specification complete. □

## 1.4 Static specification

In this section, we consider static specification. Intuitively, a static specification of a database defines the set of valid instances of the database. Valid instances are formalized using the concept of database family (set of valid states).

*Definition:* Let $\mathbf{R}$ be a database schema. A *database family* over $\mathbf{R}$ is a subset of $\text{inst}(\mathbf{R})$ which is r.e. and C-generic for some finite C.

A *static specification language* L consists of a set of words (static specifications), each of which defines a database family. Thus, for each G in L, the semantics of G is given by a database family.

In this paper, as in [AV2], we focus on static specifications induced by dynamic specifications based on admissible transactions.

We now define a completeness criteria for static specifications:

*Definition:* Let **R** be a database schema, and L a static specification language. Then L is *static specification complete* if for each database family D, there is a word in L which defines D.

Traditional static specifications are based on constraints, usually relational calculus sentences. (The database family corresponding to a set G of constraints is usually denoted by sat(G).) Note that specification languages based on constraints cannot be expected to be specification complete. Indeed, the database families defined by constraints are recursive (if the constraints can be effectively checked). However, some database families are r.e. but not recursive.

As already noted, a dynamic specification gives rise naturally to a static specification. Indeed, each transition set $\tau$ generates a database family, denoted gen($\tau$), consisting of all instances reachable from the empty instance by transitions in $\tau$. More formally, we have:

*Definition:* For each transition set $\tau$ over a database schema **R**, the *database family over* **R** *generated by* $\tau$ is

$$\text{gen}(\tau) = \{ \mathbf{I} \mid (\phi, \mathbf{I}) \in \tau^* \}$$

where $\tau^*$ is the reflexive-transitive closure of $\tau$.

With the above definition, it is clear that any dynamic specification language can also be viewed as a static specification language. If L is a dynamic specification language and T is in L, then the static specification semantics of T is given by:

$$\text{gen}_L(T) = \text{gen}(\text{tran}_L(T)).$$

The following result relates dynamic completeness and static completeness of dynamic specification languages.

*Theorem 1.4.1:* Let L be a dynamic specification language. If L is dynamic specification complete, then L is static specification complete. The converse does not hold.

*Proof:* (Sketch) Suppose that L is dynamic specification complete. Let D be a database family. Consider the set $X = \{(\phi, \mathbf{I}) \mid \mathbf{I} \in D\}$. Clearly, X is a transition set. Thus, X = tran(T) for some T in L. Clearly, $D = gen(tran_L(T)) = gen_L(T)$. Hence L is static specification complete.

To see that the converse does not hold, consider a dynamic specification language L such that:

- for each T in L, $(\mathbf{I}, \mathbf{J}) \in tran_L(t)$ implies $\mathbf{I} = \phi$;
- L is static specification complete.

Clearly, L is not dynamic specification complete. □

In our earlier discussion of dynamic specification, we identified two significant classes of transition sets: bounded transition sets, and deterministic transition sets. Each such class of transition sets induces a corresponding class of database families, as defined next.

*Definition:* A database family is *bounded* iff D = gen($\tau$) for some bounded transition set $\tau$. A database family is *deterministic* iff D = gen($\tau$) for some deterministic transition set $\tau$.

Intuitively, a database family is bounded if it can be generated using some finite set of safe procedures (possibly non-deterministic). Similarly, a database family is deterministic if it can be generated using some finite set of deterministic procedures. We now relate these classes of database families to database families defined using integrity constraints. We focus on constraints which can be expressed as sentences of first-order relational calculus. (This includes all traditional constrainsts.) We will show that some of these constraints define database families which are not bounded, and some define bounded database families which are not deterministic. However, a large class of constraints yields deterministic database families, as shown next.

*Theorem 1.4.2:* For each finite set G of EGD's and total TGD's, sat(G) is a deterministic database family.

*Proof:* The proof uses a specific deterministic transaction language (SdetTL) defined later, and is therefore provided at the end of Section 4. □

*Remark:* Theorem 1.4.2 can be extended to sets G of constraints containing certain embedded TGD's. In this case, an acyclicity condition is required of the embedded TGD's. In particular, it can be shown that the above result extends to acyclic inclusion dependencies.

Finally, we show that first-order constraints can define database families which are not bounded, and database families which are bounded but not deterministic. (In particular, this implies that the set of deterministic database families is strictly included in the set of bounded database families.) We also show that there are deterministic database families which are not describable by any first-order sentence.

*Proposition 1.4.3:*

(i) There exists a first-order sentence $\sigma$ such that sat($\sigma$) is not a bounded database family.

(ii) There exists a first-order sentence $\sigma$ such that sat($\sigma$) is a bounded but not deterministic database family.

(iii) There exists a deterministic database family D such that D $\neq$ sat($\sigma$) for each first order sentence $\sigma$.

*Proof:* (Sketch) To see (i), consider the relational schema R = ABC, and the sentence

$$\sigma = \sigma_1 \,/\backslash\, \sigma_2 \,/\backslash\, \sigma_3 \,/\backslash\, \sigma_4 \,/\backslash\, \sigma_5,$$

where

- $\sigma_1$ is the embedded join dependency

$$\forall \;\; xyzx'y'z' \; (R(xyz) \,/\backslash\, R(x'y'z') \Longrightarrow \exists \;\; z'' \; R(xy'z'')).$$

- $\sigma_2$ is the FD C $\rightarrow$ AB,
- $\sigma_3$ is the IncD R[A] $\subseteq$ R[C],
- $\sigma_4$ is the IncD R[B] $\subseteq$ R[A], and
- $\sigma_5$ is the IncD R[A] $\subseteq$ R[B].

Let I be in sat($\sigma$). Clearly, #(const(I)) = $n^2$ for some n = #($\pi_A$(I)). It can be shown that sat($\sigma$) is not bounded. (Intuitively, this is due to the sparsity of sat($\sigma$) which requires the use of an unbounded number of "new" values in state transitions within sat($\sigma$).)

Next, consider (ii). Let R and $\sigma$ be as above. Let $\xi = \sigma \;\backslash/\; \psi$, where $\psi$ corresponds to the join dependency:

$\forall\ x_1,y_1,z_1,x_2,y_2,z_2,x_3,y_3z_3\ (R(x_1,y_1,z_1)/\backslash R(x_2,y_2,z_2)/\backslash R(x_3,y_3z_3) \Longrightarrow R(x_1,y_2,z_3))$,
together with the IncD's R[A] = R[B] = R[C]. Consider sat($\xi$). To see that sat($\xi$) is bounded, let H be the transition set over inst(R)

$\{ (I,J) \mid I \models \psi,\ J \models \psi,\ \#(\mathbf{const}(J) - \mathbf{const}(I)) = 1\} \cup$

$\{ (I,J) \mid I \models \psi,\ J \models \sigma,\ J \subseteq I \}$.

Clearly, H is a bounded transition set, and gen(H) = sat($\xi$). Thus sat($\xi$) is bounded. We now show that sat($\xi$) is not deterministic. Suppose that sat($\xi$) = gen(K) for some deterministic transition set K. By definition, there exist n (n>0), some finite sets C, $C_i$ ($1 \leq i \leq n$) of constants, and deterministic ($C_i \cup C$)-generic database updates $\tau_i$ such that

$K = \cup \{ f(\tau_i) \mid 1 \leq i \leq n,\ f \text{ is an isomorphism such that } f|_C = id \}$.

Let $N = (\#(\bigcup_{i=1}^{n} C_i \cup C) + 2)^2$. Let $I \in sat(\sigma)$ be such that $\#\mathbf{const}(I) = N$. Since $sat(\sigma) \subseteq sat(\xi)$ and sat($\xi$) = gen(K), $I \in gen(K)$. By definition, there exist $\phi = I_0,\ldots,I_m = I \in sat(\xi)$ such that $(I_j,I_{j+1}) \in f_j(\tau_{i_j})$ for some isomomorphism $f_j$ such that $f_j|_C = id$. By an argument similar to that showing that sat($\sigma$) is not bounded, it can be shown that there exists k, $0 \leq k \leq m$, such that $\mathbf{const}(I_k) = \mathbf{const}(I_{k+1}) = \mathbf{const}(I)$, $I_k \models \psi$, and $I_{k+1} \models \sigma$. It follows that $\#(\pi_C(I_{k+1})) = \#(I_{k+1}) = N$. Thus, $\#(I_k) = N^3$. It follows that $\pi_{AB}(I_k)$ uses N constants, while $\pi_{AB}(I_{k+1})$ uses $\sqrt{N}$ constants. In particular, there exist $\alpha, \beta \notin \bigcup_{i=1}^{n} C_i \cup C$ such that

$$\alpha \in \mathbf{const}(\pi_{AB}(I_k)) \cap \mathbf{const}(\pi_{AB}(I_{k+1})), \text{ and}$$
$$\beta \in \mathbf{const}(\pi_{AB}(I_k)) - \mathbf{const}(\pi_{AB}(I_{k+1})).$$

(Thus $\beta$ is eliminated from $\pi_{AB}(I_k)$ in the transition to $I_{k+1}$.) Let f be the isomorphism defined by $f(\alpha) = \beta$, $f(\beta) = \alpha$, and f is the identity elsewhere. Since $I_k \models \psi$ ($I_k$ is a cross-product), it is clear that $f(I_k) = I_k$. Since $(I_k,I_{k+1}) \in f_k(\tau_{i_k})$ and $f_k(\tau_{i_k})$ is ($C_{i_k} \cup C$)-generic, it follows that $f(f_k(\tau_{i_k})) \subseteq f_k(\tau_{i_k})$, so $f((I_k,I_{k+1})) \in f_k(\tau_{i_k})$, and $(I_k,f(I_{k+1})) \in f_k(\tau_{i_k})$. Since $f_k(\tau_{i_k})$ is a mapping and $(I_k,I_{k+1}) \in f_k(\tau_{i_k})$, it follows that $f(I_{k+1}) = I_{k+1}$, so $\beta \in \mathbf{const}(\pi_{AB}(I_{k+1}))$, contradiction. Thus sat($\xi$) cannot be deterministic, which concludes the proof of (ii).

Finally, (iii) follows from Theorem 3.1 of [AV3] which implies that there exists a deterministic database family which is not recursive. □

The connection between the various classes of database families and traditional constraints

is summarized in Figure 1.4.1.

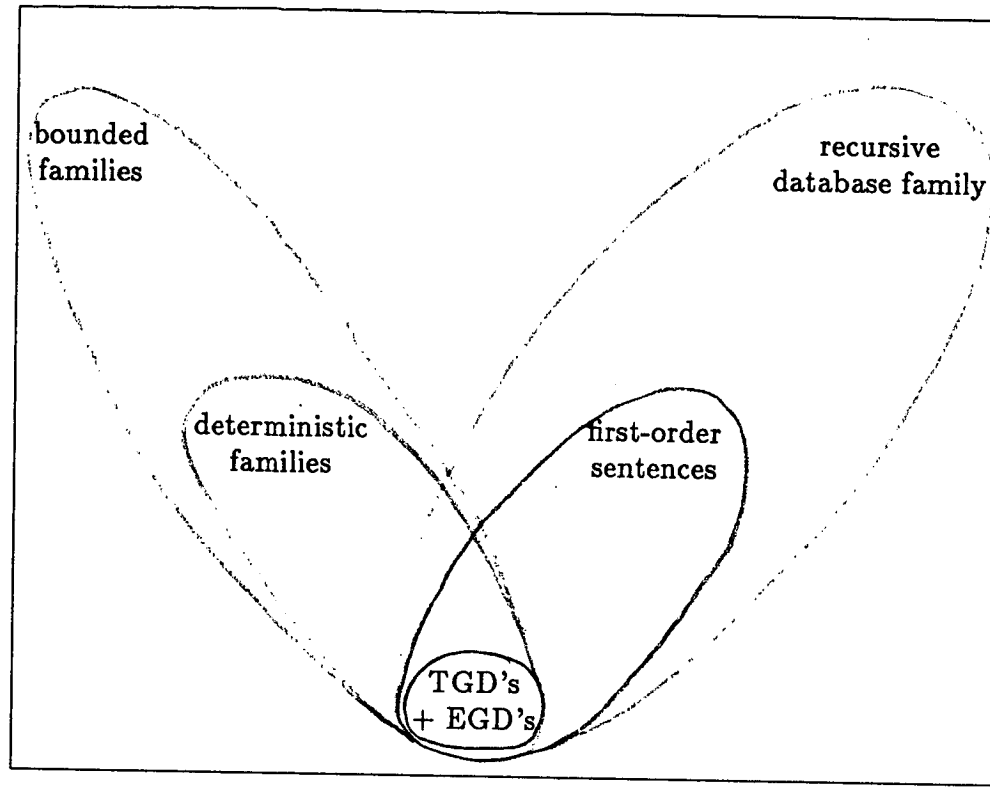


**Figure 1.4.1**

## 2. TRANSACTION LANGUAGES

In this section, we introduce the transaction languages which are used in the paper. We consider several non-deterministic languages, and their deterministic counterparts. We also study some issues related to safety, and point out a trade-off between negation in conditions and deletions.

### 2.1 Language constructs

In this section, we informally introduce the constructs used in our languages.

The basic operations allow us to insert or delete a constant tuple r in a relation R. They are denoted, respectively, by $i_R(r)$ and $d_R(r)$. The content of a relation R can be completely deleted by the operation $erase_R$.

Three constructs are used besides the basic operations: transaction composition (denoted by a ";"), a construct to perform iterations ("while"), and finally one to randomly assign a new domain value to a domain variable ("with new"). The last construct permits the introduction in the database of values which were not originally part of the active domain. (Recall that we assume an infinite domain.) Examples of transactions are now given to illustrate these three constructs:

(1) $i_R(5,6)$; $i_R(3,4)$; $i_R(5,7)$; $erase_T$,

(2) while $R(x,y) \wedge R(y,z) \wedge \neg R(x,z)$ do $i_R(x,z)$ done,

(3) $d_T(0)$; while $\neg T(0) \wedge R(x,y)$ do $i_P(x,y)$; $i_T(0)$ done,

(4) with new z do $i_T(z)$ done.

where P and R are binary relations, and T is unary.

The first transaction inserts three tuples in R and empties T. The second one closes R transitively. Intuitively, the transaction in the body of the loop is applied until the condition can no longer be satisfied. Domain variables are used in the body of a while statement, and in its condition. The semantics attached to the while is based on valuations of the variables to domain values. As we shall see, two semantics can be given to a while statement: a deterministic one and a non-deterministic one. With the non-deterministic semantics, a valuation satisfying the condition is non-deterministically chosen. The transaction in the body of the while is then applied for that valuation. This is iterated until no valuation satisfying the condition can be found. Clearly, the choice of the valuation introduces non-determinism. With the deterministic semantics, all valuations satisfying the condition are considered. The transaction of the body is applied for each such valuation. The result of one iteration is then the *union* of the results for each valuation. This is iterated until no valuation satisfying the condition can be found.

To illustrate the difference between the two semantics, consider transactions (2) and (3) defined above. Let R = {<0,1>, <1,2>, <1,3>, <2,4>}. For transaction (2), the non-deterministic semantics can lead, for instance, to the sequence of intermediate results shown in Figure 2.1.1. The sequence of intermediate results for the deterministic semantics is represented in Figure 2.1.2. Note that, in this case, the effects of the transaction are the same with the non-deterministic and deterministic semantics. This is not the case for transaction

(3). Indeed, suppose that relation P is empty when the transaction (3) is applied. After applying (3) with the deterministic semantics, P contains the same tuples as R. With the non-deterministic semantics, P contains exactly one tuple of R, chosen non-deterministically.

Transaction $t_4$ illustrates a simple use of the "with new" construct. A tuple with a new value (i.e., a value not part of the active domain) is inserted in T.

To conclude this section, we illustrate two subtleties of the "while" and "with new" constructs:

- the first involves the use of union in the deterministic semantics of the while (Example 2.1.1); and
- the second is related to the use of the "with new" in conjunction with the "while" (Example 2.1.2).

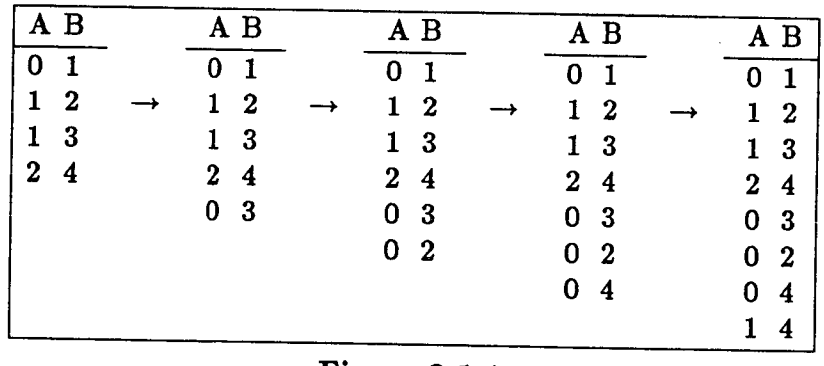


**Figure 2.1.1**

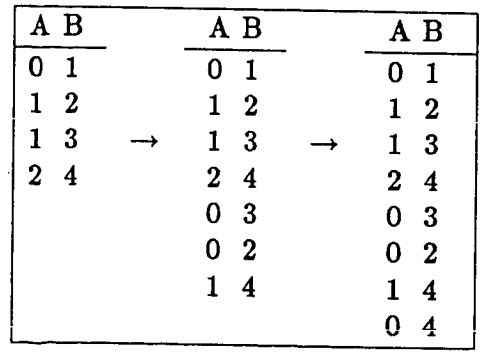


**Figure 2.1.2**

*Example 2.1.1:* Consider the transaction

$$\text{while } R(x) \text{ do } d_R(x) \text{ done.}$$

With the non-deterministic semantics of the while, the tuples of R are deleted one after another until R is empty, and the transaction terminates. With the deterministic semantics, if R has more than one tuple, the transaction never terminates. Indeed, R remains unchanged after each iteration of the loop. Intuitively, the choice of taking the union (rather than intersection) of the results after each iteration to guarantee determinism, privileges positive information, i.e., insertion.

*Example 2.1.2:* Consider the following transaction:

$\text{erase}_S$;
while $R(x,y) \wedge \neg S(x,y)$ do
with new z do $i_T(x,y,z)$;$i_S(x,y)$ done
done

Suppose that we use the deterministic semantics of the while. As mentioned above, the "with new" construct allows to "invent" new domain values. Thus, the above transaction inserts in S and T the tuples in R, marked in T with invented values. In the case of a deterministic while, we shall assume that two distinct branches of the while (that can be viewed as realized in parallel) can not invent the same value. Suppose that the relation R is as in Figure 2.1.3. A possible state of T and S after a run of the transaction is shown in Figure 2.1.3. Note that the transaction has infinitely many possible distinct outcomes which are isomorphic, but differ in the particular choice of new values. Some syntactic restrictions can be imposed to guarantee that the result does not contain any of the new values. Under those restrictions, and with a deterministic semantics of the while, we will obtain a deterministic language.

| R | A | B |
|---|---|---|
| | 0 | 1 |
| | 0 | 5 |
| | 0 | 4 |
| | 0 | 2 |

→

| T | A | B | C |
|---|---|---|---|
| | 0 | 1 | 9 |
| | 0 | 5 | 8 |
| | 0 | 4 | 3 |
| | 0 | 2 | 11 |

| S | A | B |
|---|---|---|
| | 0 | 1 |
| | 0 | 5 |
| | 0 | 4 |
| | 0 | 2 |

**Figure 2.1.3**

## 2.2 The transaction language TL

In order to formally present the language, we need the auxiliary concepts of atomic formula,

and condition.

*Definition*: Let **R** be a database schema. For each R in **R**, and each tuple r (possibly with variables) over R, R(r), and $\neg$ R(r) are *atomic formulas over* **R**. For each x in **var**, and y in **var** $\cup$ **dom**, x = y and x $\neq$ y are *atomic formulas over* **R**. Nothing else is an atomic formula. A *condition* over a database schema **R** is an expression $Q_1 /\backslash \cdots /\backslash Q_n$ where each $Q_i$ is an atomic formula over **R**. The set of variables in a condition Q is denoted by **var** (Q).

Let Q be a condition with **var**(Q) = $\phi$. The fact that Q is *satisfied* by an instance **I** is defined in the obvious fashion, and is denoted by **I** $\models$Q.

We now present the notions of parameterized transaction and transaction. Informally, parameterized transactions are transactions with "free" variables (not bound to any condition), which are used in a manner analogous to procedure parameters. Free variables are formally defined below concomitantly with transactions:

*Definition:* a *parametrized transaction (p-transaction)* in **TL** over a database schema **R** is an expression obtained as follows.

(1) for R in **R**, and for each tuple r over R, $i_R(r)$, $d_R(r)$, and $erase_R$ are p-transactions (respectively called *insertion, deletion* and *erase*), **free**($erase_R$) = $\phi$, and **free**($i_R(r)$) = **free**($d_R(r)$) is the set of variables appearing in r;

(2) if t, and t' are p-transactions over **R**, then (t;t') is a p-transaction over **R**, and **free**((t;t'))=**free**(t) $\cup$ **free**(t');

(3) if t is a p-transaction and Q a condition over **R**, then

t'$\equiv$ while Q do t done

is a p-transaction over **R**, and **free**(t')=**free**(t)-**var**(Q); and

(4) if t is a p-transaction and z is a free variable in t, then

t'$\equiv$ with new z do t done

is a p-transaction over **R**, and **free**(t')=**free**(t) - {z}.

A *transaction* t in (**TL**) is a p-transaction with **free**(t)= $\phi$.

We occasionally use convenient abbreviations. For instance, "with new $z_1,\ldots,z_n$ do t done" will be used instead of

```
with new z_1 do
  with new z_2 do
    ...
        with new z_n do
           t
           done
        done
    ...
  done
```

Other abbreviations like "if-then-else" or "case" statements will be defined when needed.

We shall also use the following.

*Notation:* The set of relational schemas occurring in a transaction t constitutes a database schema denoted by **sch**(t). The set of constants occurring in a transaction t is denoted by **const**(t). The set **const**(t) ∪ **const**(**I**) is denoted by **const**(**I**,t).

In order to define the semantics of transactions, we need the auxiliary concept of valuation of p-transactions and conditions.

*Definition:* Let t be a p-transaction. Let X be a set of free variables in t. A *valuation* v of t (on X) is a mapping from X into **dom**. A valuation v of t transforms t into vt obtained by replacing in t each free variable x in X by v(x). The *valuation* of a condition is defined similarly.

Note that, if X = **free**(t) and v is a valuation of t on X, then vt is a transaction.

The semantics of a transaction is now given in terms of the effect of the transaction on each database instance (in the spirit of [AV1]). As already mentioned, the effect of a transaction on a given database is not always uniquely determined. Therefore, the effect is defined as a binary relation between database instances (rather than a mapping).

*Definition:* The *effect* of a transaction t in TL, denoted by $\text{eff}_{\text{TL}}(t)$, or eff(t) when it is understood that t is in TL, is a binary relation defined on **inst**(**sch**(t)) × **inst**(**sch**(t)) as follows:

(1-a) for R in **R**, and for each constant tuple r over R, $(\mathbf{I},\mathbf{J}) \in \text{eff}(i_R(r))$ iff
$\mathbf{J}(R) = \mathbf{I}(R) \cup \{r\}$, and $\mathbf{J}(S) = \mathbf{I}(S)$ for each $S \neq R$;

(1-b) for R in **R**, and for each constant tuple r over R, $(\mathbf{I}, \mathbf{J}) \in \text{eff}(d_R(r))$ iff
$\mathbf{J}(R) = \mathbf{I}(R) - \{r\}$, and $\mathbf{J}(S) = \mathbf{I}(S)$ for each $S \neq R$;

(1-b) for R in **R**, $(\mathbf{I}, \mathbf{J}) \in \text{eff}(\text{erase}_R)$ iff
$\mathbf{J}(R) = \phi$, and $\mathbf{J}(S) = \mathbf{I}(S)$ for each $S \neq R$;

(2) if t;t' is a transaction over **R**, then $(\mathbf{I},\mathbf{J}) \in \text{eff}(t;t')$ iff there exists **K** such that $(\mathbf{I},\mathbf{K}) \in \text{eff}(t)$ and $(\mathbf{K},\mathbf{J}) \in \text{eff}(t')$;

(3) if t' $\equiv$ while Q do t done is a transaction over **R**, then $(\mathbf{I},\mathbf{J}) \in \text{eff}(t')$ iff there exist $n \geq 0$, $\mathbf{I} = \mathbf{J}_0, \mathbf{J}_1, \ldots, \mathbf{J}_n = \mathbf{J}$, and some valuations $v_0, \ldots, v_{n-1}$ of **var**(Q) such that
(3-a) for each i $(0 \leq i < n)$, $v_i$ is a valuation with values in $\mathbf{const}(\mathbf{J}_i, t')$,
$\mathbf{J}_i \models v_i Q$, and $(\mathbf{J}_i, \mathbf{J}_{i+1}) \in \text{eff}(v_i t)$;
(3-b) for each valuation v, $\mathbf{J} \not\models vQ$;

(4) if t' $\equiv$ with new z do t done is a transaction over **R**, then eff(t') =
$\{ (\mathbf{I},\mathbf{J}) \mid (\mathbf{I},\mathbf{J}) \in \text{eff}(vt)$ for some valuation v of z, $vz \notin \mathbf{const}(\mathbf{I},t') \}$.

We now make a brief comment on the semantics of the "with new" construct. Consider the transaction

$$t = \text{with new z do } i_R(z) \text{ done; } d_R(5)$$

According to (4), the value 5 may be assigned to z. This is due to the purely local interpretation chosen for "with new". Specifically, the value assigned to z is not in the active domain at the particular instant when "with new" is applied. Clearly, it is possible to choose a different semantics, more globally oriented. For instance, one may require that the new value does not occur in the global transaction. Note that it is possible to enforce that semantics with the given TL semantics, by introducing a particular relation, say ACT_DOM, where one inserts first the values occurring in the database or in the transaction, and then updates ACT_DOM each time a value is introduced by a "with new" statement.

The proof of the following result is straightforward:

*Proposition 2.2.1:* Let t be a TL transaction. Then eff(t) is C-generic, where $C = \mathbf{const}(t)$. □

When a transaction is applied to a given database, its effect is often interpreted by identifying some relations as input relations, and other relations as output relations. If the transaction is viewed as an update, the output schema is usually identical to the input schema; if the transaction is viewed as a query, the output schema usually differs from the input schema. We do not distinguish here between these two cases, but rather treat them in a uniform manner. In addition to semantically significant input and output relations, the transactions may use "temporary" relations. Thus, it appears useful to also define the effect of a transaction with respect to specified input and output database schemas. We first define the notion of input-output (i-o) schema.

*Definition:* An *input-output (i-o) schema* is a pair $<\mathbf{R},\mathbf{S}>$, where $\mathbf{R}$ and $\mathbf{S}$ are database schemas called the *input schema* and the *output schema*, respectively.

We now define the effect of a transaction with respect to an i-o schema:

*Definition:* Let t be a transaction and $<\mathbf{R},\mathbf{S}>$ an i-o schema such that $\mathbf{R} \cup \mathbf{S} \subseteq \mathbf{sch}(t)$. The effect of t with respect to $<\mathbf{R},\mathbf{S}>$, denoted $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t)$, is the subset of $\mathbf{inst}(\mathbf{R}) \times \mathbf{inst}(\mathbf{S})$:

$$\{<\mathbf{I},\mathbf{J}> \mid \exists\ <\mathbf{I}',\mathbf{J}'> \in \text{eff}(t),\ \mathbf{I}'|_{\mathbf{R}} = \mathbf{I},\ \mathbf{I}'|_{\mathbf{sch}(t)-\mathbf{R}} = \phi, \text{ and } \mathbf{J}'|_{\mathbf{S}} = \mathbf{J}\}.$$

For two transactions t and t' over an i-o schema $<\mathbf{R},\mathbf{S}>$, if $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t) = \text{eff}_{<\mathbf{R},\mathbf{S}>}(t')$, we say that they are $<\mathbf{R},\mathbf{S}>$-*equivalent.*

From the definition, relations which are not in the input schema are assumed to be empty before the transaction is executed. After the transaction is run, the relations in the output schema must contain the desired result. (The content of the other relations is immaterial.) Note that the C-genericity of eff(t) implies the C-genericity of $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t)$ for each $\mathbf{R}$ and $\mathbf{S}$.

We now illustrate these various concepts.

*Example:* Let $\mathbf{R}$= {Manager(AB)} and $\mathbf{S}$= {S(C)}. Each of the following transactions over the i-o schema $<\mathbf{R},\mathbf{S}>$ computes a query described in parentheses:

- (compute in S the names of all employees managed by Jeremie):
  while Manager("Jeremie",y) /\ ¬ S(y) do $i_S(y)$ done
- (compute in S the names of all persons who have more employees than managers).
  while Manager(x,y) do $i_T(x,y);i_{T'}(y,x)$ done;

while T(x,y) /\ T'(x,z) do $d_T(x,y);d_{T'}(x,z)$ done;
while T(x,y) do $i_S(x);d_T(x,y)$ done.

Note that the second transaction uses temporary relations T and T'.

*Example:* Let **R**= **S**= {Manager(AB)}. The following transactions over the i-o schema <**R**,**S**> realize certain updates, described in parentheses.

- (delete all tuples with A=a or B=b from R)
  while R(a,y) do $d_R(a,y)$ done; while R(x,b) do $d_R(x,b)$ done.
- (modify all tuples with A=a to A=a' in R)
  while R(a,y) do $d_R(a,y);i_R(a',y)$ done.

It should be noted that the set of constructs in TL is not minimal. Clearly, deletions (respectively, erase) can be simulated using the other constructs. However, the sets {while, with, insert, deletion, ";"} and {while, with, insert, erase, ";"} are minimal, and yield the same power as the full language. Furthermore, we will consider in a forthcoming section a trade-off between negation in conditions and deletes.

To conclude this section, we briefly consider two alternatives for the TL language:

($\alpha$) It is possible to consider a fixpoint semantics of the "while": "while" statements are interrupted when no iteration can modify the database state. This does not change the power of the language.

($\beta$) Let TL' be the language obtained by allowing any first order relational calculus formula in the condition of a "while" statement. The language thereby obtained has exactly the power of TL.

## 2.3 Safe Transactions

As seen in the previous section, our definition of the language TL allows transactions to produce domain values not present in the original database or in the transaction, and which are thus "unsafe". In this section, we motivate the choice of allowing unsafe transactions and compare it with the approach of Chandra and Harel. Then we exhibit two simple syntactic restrictions which guarantee safety. We investigate the different notions of safety, and present safe variations of TL.

The safety conditions usually required in the database context is violated by the fact that transactions may require the invention of domain values not found in the current database state. However, in order to achieve completeness in terms of computability, it is necessary to simulate the computations of a wide class of partially recursive functions (i.e., the class of deterministic updates). This is not possible by using a fixed database schema and domain values from the input database alone. The approach adopted in [CH1] is essentially to simulate counters using the number of attributes of a relation. Thus, an unbounded number of new relational schemas are produced in the course of the computation. On the other hand, our approach is to use a fixed database schema, and simulate counters using chains of "invented" domain values, not necessarily present in the input database. We believe that the latter approach is more natural in a database context. Indeed, while generally it seems inappropriate to accept arbitrary, invented values in the final result, it may be useful to allow invented values throughout the computation, as shown in the following example (where invented values are only used in temporary relations).

*Example:* Consider the transaction of Figure 2.3.1. This transaction uses a base relation P(arent), and computes the same generation cousins in relation SGC. A tuple $(x,y,\alpha)$ in the relation "generation" indicates that x is an ancestor of y of level $\alpha$. Invented values are

```
with new γ do
  while P(x,y) /\ ¬ generation(x,y,γ) do
    i_generation(x,y,γ) done;
  i_current(γ);
  done
i_notend(1);
while notend(1) /\ current(α) do
  with new β do
    d_notend(1);
    while P(x,y) /\ generation(y,z,α) /\ ¬ generation(x,z,β)
      do i_generation(x,z,β); i_notend(1)  done;
    d_current(α); i_current(β)
    done
  done
while generation(x,y,z) /\ generation(x,y',z) /\ ¬SGC(y,y') do
  i_SGC(y,y') done
```

**Figure 2.3.1: Same Generation Cousin**

introduced in the relations current, and generation, and are used to name generations. If there is no data cycle in the parent relation, the transaction terminates, and computes the correct answer. Note that this computation can be achieved by simpler transactions, and this example only illustrates one possible technique.

We now present the central concept of the section: safety.

*Definition:* A transaction t is *safe* with respect to an i-o schema <**R**,**S**> iff for each (**I**,**J**) in $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t)$, the set of constants appearing in **J** is included in those of **I** together with those of t.

We now present restrictions of TL which are safe. A first restriction that can be imposed is to forbid the invention of values. This yields *strongly safe TL* (STL), the language obtained from TL by removing the "with new" construct. This strong safety condition is usually the condition required in query languages. However, as mentioned earlier, this severely limits the computational power of the language. An alternative is to allow the use of invented values in temporary relations, but not in relations of the i-o schema. This can be done by imposing syntactic restrictions. To this end, we use the following auxiliary concept:

*Definition:* Let t be a transaction. An occurrence $\chi$ of a variable x in t is *positively bound* by a relation R iff for some $Q_i$, and t'

(a) t≡ ... while $Q_1 \wedge \ldots \wedge Q_n$ do t' done ...,

(b) $\chi$ is in t', and

(c) for some i, r, and A, $Q_i \equiv R(r)$ and r(A)=x.

We next exhibit a weaker version of safety, where the invention of new values is tolerated just in temporary relations.

*Definition:* Let t be a transaction over the i-o schema <**R**,**S**> Then t is *weakly safe* (w.r.t. <**R**,**S**>) iff for each R in $\mathbf{R} \cup \mathbf{S}$ and each occurrence $\chi$ of a variable in some insertion in R, $\chi$ is positively bound by some relation in $\mathbf{R} \cup \mathbf{S}$. This syntactic restriction leads to a variation of TL, called *weakly safe TL* (WTL).

The following result relates the various notions of safety.

*Theorem 2.3.1:* In TL,

(i) Any strongly safe transaction is weakly safe (i.e., STL $\subseteq$ WTL).

(ii) Any weakly safe transaction is safe.

(iii) The converses of (i) and (ii) do not hold.

(iv) For any safe transaction, there exists an equivalent weakly safe transaction.

(v) There exists a weakly safe transaction with no equivalent strongly safe transaction.

*Proof:* (i) and (ii) are obvious. Consider (iii). Let S and T be unary relations, and s and t be transactions defined as follows:

$s$ = with new z do $i_T(z)$ done

$t$ = with new z do $i_S(z)$; $d_S(z)$ done

It is easily seen that s is weakly safe w.r.t. <{S},{S}>, and is not strongly safe. Also, t is safe w.r.t. <{S},{S}>, but is not weakly safe w.r.t. <{S},{S}>.

Consider (iv). Let t be a safe transaction w.r.t. an i-o schema $<\mathbf{R},\mathbf{S}>$. For each R in $\mathbf{R} \cup \mathbf{S}$, let R' be a new relation. We construct a transaction t' which is weakly safe and has the same effect as t. Let

$$t' = t_{copy}; t_{simul}; t_{insert}; t_{delete}$$

where

- $t_{copy}$ copies each R in $\mathbf{R}$ into R';
- $t_{simul}$ simulates the transaction t on the copies of relations;
- $t_{insert}$ inserts, for each R in $\mathbf{R} \cup \mathbf{S}$, each tuple of R' into R;
- $t_{delete}$ deletes, for each R in $\mathbf{R} \cup \mathbf{S}$, the tuples of R that do not occur in R'.

Since $t_{copy}$, $t_{simul}$, and $t_{delete}$ do not insert in $\mathbf{R} \cup \mathbf{S}$, it is clear that they can be expressed in WTL. Consider next $t_{insert}$. For simplicity, consider first the case when $\mathbf{const}(t) = \phi$. Let n be the maximal arity of a relation in $\mathbf{R} \cup \mathbf{S}$, and $\{x_i \mid i \text{ in } [1..n]\}$, $\{x_{i,j} \mid i,j \text{ in } [1..n]\}$ be two sets of disjoint variables. The transaction $t_{insert}$ is the concatenation of all transactions of the form

while $R'(x_1,...,x_m) \wedge R_1(u_1) \wedge ... \wedge R_m(u_m) \wedge \neg R(x_1,...,x_m)$ do
$i_R(x_1,...,x_m)$
done

where $R \in \mathbf{R} \cup \mathbf{S}$, arity(R) = m, and, for each i, $1 \leq i \leq m$,

(a) $R_i \in \mathbf{R} \cup \mathbf{S}$,

(b) for each j, $1 \leq j \leq$ arity($R_i$), the j-th component of $u_i$ is either $x_i$ or $x_{i,j}$, and

(c) for some j, $1 \leq j \leq$ arity($R_i$), the j-th component of $u_i$ is $x_i$.

By (c), $t_{insert}$ is weakly safe. If $\mathbf{const}(t) \neq \phi$, then the construction of $t_{insert}$ is slightly modified to allow (in tuples inserted in $\mathbf{R} \cup \mathbf{S}$) constants in $\mathbf{const}(t)$ in addition to variables bound to $\mathbf{R} \cup \mathbf{S}$. It is now easy to see that, since t is safe w.r.t. $<\mathbf{R},\mathbf{S}>$, t and t' are $<\mathbf{R},\mathbf{S}>$-equivalent.

Consider (v). It will be shown in Section 3 that weakly safe and strongly safe transactions compute different classes of database updates. This implies that (v) holds. □

It should be noted that query safety as in [U] corresponds to strong safety in the present paper. Indeed, for each safe (according to [U]) first order relational query, there exists a strongly safe transaction which computes that query. (The converse does not hold.) Finally, note that (iv) provides an important connection between semantic safety and syntactic safety. This is analogous to the connection established in [M] between safe queries and queries whose limited and unlimited evaluations are identical.

### 2.4 Trade-off Between Negation and Deletion

The TL-languages have few constructs. We already mentioned that the set {with, insert, delete, while, ";"} is a minimal set of constructs which yields the same power as all of TL. In this section, we exhibit a trade-off between negation in conditions and deletions which shows that (1) deletion can essentially be replaced by negation in conditions, and (2) negation in conditions can be replaced by deletions.

The first theorem states that there is essentially no need for deletion if the input and output schemas are disjoint. In that case (but not in general), deletions can be avoided by using appropriate negations in conditions.

*Theorem 2.4.1* Let t be a TL-transaction over an i-o schema <**R**,**S**> with $\mathbf{R} \cap \mathbf{S} = \phi$. Then there is an $<\mathbf{R},\mathbf{S}>$-equivalent TL-transaction t' without deletion and erase.

*Proof:* To simplify the presentation, we assume that t uses only one input relation (R), and one output one (S). The general case is treated similarly. Intuitively, time-stamps are used. Two new relations R' and S' are used. A tuple in R' will consist of a tuple in R together with a time-stamp, similarly for S and S'. The unary relation OLD is used to keep the outdated time-stamps. A tuple in {R',S'} with time-stamp $\alpha$ represents a tuple in the corresponding state of {R,S} if $\alpha$ does not belong to OLD. The transaction t has the same effect for the schema <R,S> as

$$t' = t_{copy}; t_{simul}; t_{recopy}$$

where

- $t_{copy}$ time-stamps the tuples in R, and copies them into R' using a temporary relation T:
  while R(u) /\ ¬ T(u) do
  $i_T(u)$;
  with new x do $i_{R'}(u,x)$ done
  done.
- $t_{simul}$ simulates the computation of t in R' and S' and handles the time-stamps. The transaction $t_{simul}$ is obtained from t by:
  (i) replacing each condition R(u) by R'(u,x) /\ ¬ OLD(x);
  (ii) replacing an insertion $i_R(u)$ in R by
  with new x do $i_{R'}(u,x)$ done
  (iii) replacing a deletion $d_R(u)$ in R by
  while R'(u,x) /\ ¬ OLD(x) do $i_{OLD}(x)$ done
  (iv) insertions and deletions in S are treated similarly.
- $t_{recopy}$ copies in S all tuples u such that <u,$\alpha$> is in S', and $\alpha$ is not outdated:
  while S'(u,x) /\ ¬ OLD(x) /\ ¬ S(u) do $i_S(u)$ done

It is easy to check that t' uses no deletion or erase, and that t and t' are <R,S>-equivalent. □

Clearly, delete and erase cannot be both eliminated if $\mathbf{R} \cap \mathbf{S} \neq \phi$. However, using a similar construction, one can show that, in that case, one erase for each relation in **R** is always

sufficient. More precisely, for each transaction t, and each i-o schema $<\mathbf{R},\mathbf{S}>$, there is an $<\mathbf{R},\mathbf{S}>$-equivalent transaction

$$t' = t_{copy};\ erase_{R_1};\ldots\ erase_{R_n};\ t_{simul};\ t_{recopy}$$

where $\mathbf{R} \cap \mathbf{S} = \{R_1,\ldots,R_n\}$, and $t_{copy}$, $t_{simul}$ and $t_{recopy}$ are as in the previous construction, and thus do not contain any delete or erase operation.

The second result of this section shows that negation in conditions can be replaced by appropriate deletions.

*Theorem 2.4.2:* Let t be a transaction in TL over an i-o schema $<\mathbf{R},\mathbf{S}>$. Then there is an $<\mathbf{R},\mathbf{S}>$-equivalent transaction pos(t) in TL with no negative literals in conditions.

*Proof:* Intuitively, for each relation R used by t, a relation R' is used which stores the complement of R w.r.t. the active domain. The simulation of the computation is complicated by the fact that the active domain changes throughout the computation due to (a) the invention of values ("with new"), and (b) the removal of values from the active domain due to tuple deletions.

Before defining pos(t), we show how the relations R' can be computed by a transaction $t_{comp}$. Let $\mathbf{I}$ be the current state. The transaction $t_{comp}$

(1) computes the active domain DOM = $\mathbf{const}(t,\mathbf{I})$; then

(2) for each R, computes the complement of R, i.e., $DOM^{arity(R)}$ - R, in R'.

For (1), first note that for each R, a copy S of R can be obtained without modifying R using an auxiliary relation T, as follows:

while R(u) do $d_R(u)$; $i_T(u)$ done;

while T(u) do $d_T(u)$; $i_R(u)$; $i_S(u)$ done.

Using copies of relations, $\mathbf{const}(\mathbf{I})$ can be computed in DOM. Adding the constants in t is straightforward.

Now consider (2). For each i, $DOM^i$ can be computed inductively. We show how to obtain $DOM^{i+1}$ from $DOM^i$. Suppose that $DOM^i$ is in a relation V. Let X and Y be relations of arity i (empty at the beginning), and Z be a copy of DOM. Then $DOM^{i+1}$ is computed in

W as follows:

while V(v) do $i_X(v)$; $i_Y(v)$; $d_V(v)$ done;
while Z(x) do
    while X(v) do $i_W(x,v)$; $d_X(v)$ done;
    while Y(v) do $i_V(v)$; $d_Y(v)$ done;
    while V(v) do $i_X(v)$; $i_Y(v)$; $d_V(v)$ done;
    $d_Z(x)$
done.

With the $DOM^i$ computed, it is now straightforward to obtain R':

(1) copy R in some temporary relation S,

(2) copy $DOM^{arity(R)}$ in R', and

(3) while S(u) do $d_{R'}(u)$; $d_S(u)$ done.

Finally, pos(t) is defined recursively by:

- pos(t) = t if t is a deletion, insertion or erase;
- pos(t;t') = pos(t); pos(t');
- pos(with new z do t done) = with new z do pos(t) done; and
- pos(while Q do s done) =

$$t_{empty}; t_{comp}; \text{while } Q' \text{ do pos(s)}; t_{empty}; t_{comp} \text{ done}$$

where Q' is obtained from Q by replacing each occurrence of a negative literal $\neg$ R(r) by R'(r), and $t_{empty}$ empties each R' and the temporary relations used in $t_{comp}$.

It is easily seen that, by construction, t and pos(t) are $<\mathbf{R},\mathbf{S}>$-equivalent.

□

## 2.5 Deterministic TL

We now consider a deterministic version of TL.

As mentioned above, the TL transactions may be non-deterministic for two reasons:

- they introduce values which are arbitrary, and
- the choice of a valuation in a while loop is arbitrary.

In fact, non-determinism may also arise for a third reason: a new value randomly chosen locally may coincide with a constant used later in the transaction. In that case, the choice of that value may affect the computation. We are going to use a different semantics of the while. Furthermore, we enforce constraints that guarantee that an invented value is not a constant occurring in the transaction. With these constraints, only the presence of invented values in the result may cause non-determinism. Therefore, we are going to also require weak safety. (Recall that in weakly safe transactions, the invented values cannot appear in the result.) We will see that this is sufficient to guarantee determinism w.r.t. the final result.

We now define the language detTL. Let $<\mathbf{R},\mathbf{S}>$ be an i-o schema. A detTL transaction is a weakly safe transaction over the i-o schema $<\mathbf{R},\mathbf{S}>$. (Syntactically, there is no difference between WTL and detTL). To define the semantics of detTL transactions, we use the following:

*Notation:* For each instances **I**, **J**, and each transaction t,

$$\mathbf{new(I,t,J)} = \mathbf{const(J)} - \mathbf{const(I,t)}$$

Intuitively, if t transforms **I** into **J**, new(**I**,t,**J**) is the set of constants in **J** that were "invented" during the computation.

Now we have:

*Definition:* The *effect* of a transaction t in detTL, denoted by $\mathrm{eff}_{\mathrm{detTL}}(t)$, or eff(t) when it is understood that t is in detTL, is a binary relation defined on $\mathbf{inst}(\mathrm{sch}(t)) \times \mathbf{inst}(\mathrm{sch}(t))$ as follows:

(1-a) for R in **R**, and for each constant tuple r over R, $(\mathbf{I},\mathbf{J}) \in \mathrm{eff}(i_R(r))$ iff $\mathbf{J}(R) = \mathbf{I}(R) \cup \{r\}$, and $\mathbf{J}(S) = \mathbf{I}(S)$ for each $S \neq R$;

(1-b) for R in **R**, and for each constant tuple r over R, $(\mathbf{I},\mathbf{J}) \in \mathrm{eff}(d_R(r))$ iff $\mathbf{J}(R) = \mathbf{I}(R) - \{r\}$, and $\mathbf{J}(S) = \mathbf{I}(S)$ for each $S \neq R$;

(1-c) for R in **R**, $(\mathbf{I}, \mathbf{J}) \in \mathrm{eff}(\mathrm{erase}_R)$ iff $\mathbf{J}(R) = \phi$, and $\mathbf{J}(S) = \mathbf{I}(S)$ for each $S \neq R$;

(2) if t;t' is a transaction over $\mathbf{R}$, then $(\mathbf{I},\mathbf{J}) \in \text{eff}(t;t')$ iff there exists $\mathbf{K}$ such that $(\mathbf{I},\mathbf{K}) \in \text{eff}(t)$, $(\mathbf{K},\mathbf{J}) \in \text{eff}(t')$, and

$$(*)\ \text{new}(\mathbf{I},t,\mathbf{K}) \cap \mathbf{const}(t') = \phi.$$

(3) if $t' \equiv$ while Q do t done is a transaction over $\mathbf{R}$, then $(\mathbf{I},\mathbf{J}) \in \text{eff}(t')$ iff there exist $n \geq 0$, $\mathbf{I} = \mathbf{J}_0, \mathbf{J}_1, \ldots, \mathbf{J}_n = \mathbf{J}$, such that

(3-a) for each i, $\mathbf{J}_{i+1} = \bigcup_{v \in \mathbf{V}} \mathbf{J}_{i,v}$ where V is the (finite) set of valuations of Q with values in $\mathbf{const}(\mathbf{J}_i,t')$ such that $\mathbf{J}_i \models vQ$, and $(\mathbf{J}_i, \mathbf{J}_{i,v}) \in \text{eff}(vt)$ for each v;

(3-b) for each i, and valuations v,w ($v \neq w$),

$$(**)\ \text{new}(\mathbf{J}_i,t',\mathbf{J}_{i,v}) \cap \text{new}(\mathbf{J}_i,t',\mathbf{J}_{i,w}) = \phi.$$

(3-c) for each valuation v, $\mathbf{J} \not\models vQ$;

(4) if $t' \equiv$ with new z do t done is a transaction over $\mathbf{R}$, then $\text{eff}(t') =$ $\{ (\mathbf{I},\mathbf{J}) \mid (\mathbf{I},\mathbf{J}) \in \text{eff}(vt)$ for some valuation v of z, $vz \notin \mathbf{const}(\mathbf{I},t)\}$.

Note that (*) guarantees that a transaction chooses locally invented values which do not occur in the remainder of the transaction. Note also that, by (**), two branches of a while which are executed in parallel can not choose the same new value.

It is easy to see that the effect of a detTL transaction t is C-generic, where $C = \mathbf{const}(t)$. Furthermore, as shown in the next result, detTL transactions are deterministic in the sense that applying a detTL transaction to a given instance yields at most one instance of the output schema. (Note that a detTL transaction may loop forever.)

*Proposition 2.5.1:* Let t be a detTL-transaction over some i-o schema $<\mathbf{R},\mathbf{S}>$. Then $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t)$ is a mapping.

*Proof:* (Sketch) Consider two computations of the same transaction t on the same instance $\mathbf{I}$ of $\mathbf{R}$. Using (*) and (**), it can be shown that there is a bijection f on D which is the identity on $\mathbf{const}(\mathbf{I},t)$ such that the result of the second computation is an image under f of the result of the first computation. Due to weak safety, the two computations yield instances over $\mathbf{S}$ which contain only constants in $\mathbf{const}(\mathbf{I},t)$. It follows that the results are identical. □

As we shall see, for each detTL transaction, there is a (W)TL transaction which has the same effect. By Proposition 2.5.1, the reverse is of course not true.

As in the TL case, we can require strong safety of transactions in detTL. The language thereby obtained from detTL by disallowing the "with new" construct is called *strongly safe deterministic TL*, denoted SdetTL.

The particularities of languages introduced so far are summarized in Figure 2.5.1. In the next sections, we characterize the power of these languages w.r.t. updating and specification.

| | non-deterministic semantics | deterministic semantics |
|---|---|---|
| arbitrary invented values | TL | |
| invented values in temporary relations only | WTL | detTL |
| no invented values | STL | SdetTL |

**Figure 2.5.1**

## 3. UPDATE COMPLETENESS

In this section, we prove the update completeness, respectively, the deterministic update completeness of the languages TL and detTL. We also characterize the classes of updates which can be computed using the languages WTL, STL, and SdetTL.

### *3.1 Update completeness of TL*

We show here that TL is non-deterministic update complete. By the definition of Section 1, we have to show that, for every i-o schema $<\mathbf{R},\mathbf{S}>$ and binary relation $\rho$ on $\mathrm{inst}(\mathbf{R})\times\mathrm{inst}(\mathbf{S})$ which is r.e. and C-generic, there exists a transaction t in TL such that $\mathrm{eff}_{<\mathbf{R},\mathbf{S}>}(t) = \rho$. The proof consists of two main parts:

(i) show that TL can simulate counters, and thus has full Turing Machine capability, and

(ii) show that TL can be used to compute the "Godel number" of an instance of $\mathbf{R}$ and, conversely, that an instance of $\mathbf{S}$ can be computed from its Godel number.

Before proceeding with the proof, we note that the following constructs (whose semantics are the obvious ones) can be simulated using TL:

1) if <condition> then do s done:
   $d_T(0)$;
   while <condition> /\ ¬ T(0) do s ; $i_T(0)$ done.

2) $R \overset{+}{\leftarrow} Q$ (append Q to R):
   while Q(x) /\ ¬ R(x) do $i_R(x)$ done

3) $R \overset{+}{\leftarrow} \Pi_A(Q)$ (append to R the projection of Q on an attribute A, where R is a unary relation): similar to (2).

We also use the following meta-construct:

for each i ∈ F do statement(i) done

where F is a finite set. The above denotes a concatenation of all "statement(i)", i ∈ F, in some arbitrary order.

Let $\rho$ be an r.e. subset of $\mathbf{inst(R)} \times \mathbf{inst(S)}$, which is C-generic for some finite $C = \{c_1,...,c_k\}$. The constructions for (i) and (ii) are based on "simulating" the integers using the constants in the database, those in C, and invented values. For instance, the integers {1,2,...,n} will be represented by constants $\{a_1,...,a_n\}$ using a binary relation $R_N$ as in Figure 3.1.1. Once $R_N$ is constructed, the constant $a_i$ is used to denote integer i (zero is represented by ¢). If, in the course of the computation, it is necessary to use an integer larger than n, the relation $R_N$ is extended using an invented value. The constants $a_i$ used in $R_N$ are chosen non-deterministically, except for the constants $c_i$ in C, which are assigned deterministically to the integers {1,...,k}. We now elaborate further upon the construction of $R_N$. We start by showing how the set of integers represented at a given time by $R_N$ can be extended by one, using a

| A | B |
|---|---|
| ¢ | $a_1$ |
| $a_1$ | $a_2$ |
| $a_2$ | $a_3$ |
| ... | ... |
| $a_{n-1}$ | $a_n$ |
| $a_n$ | $ |

**Figure 3.1.1: $R_N$**

value $a_{n+1}$:

*extend*($a_{n+1}$):

If $R_N(x\$)$ then do $d_{R_N}(x\$)$; $i_{R_N}(xa_{n+1})$; $i_{R_N}(a_{n+1}\$)$ done.

The construction of $R_N$ proceeds as follows:

1) Let $C = \{c_1,...,c_k\}$. The constants in C are first placed in $R_N$ using a sequence of insertions:

$i_{R_N}(\¢ c_1)$; $i_{R_N}(c_1c_2)$;...$i_{R_N}(c_k\$)$.

1) Place all constants in the database in a unary relation D, using append statements of the form $D \overset{+}{\leftarrow} \Pi_A(S)$, for each relation S in the database and attribute A of S.

2) (a) The constants which occur in both C and D are first removed from D; and
(b) the constants still in D are next placed in $R_N$:

while D(x) do extend(x); $d_D(x)$ done

4) If, in the course of the computation, $R_N$ represents the integers up to n, and n+1 is required, $R_N$ is extended by one, using an invented value:

with new $\alpha$ do extend($\alpha$) done.

The relation $R_N$ represents an ordering of a set of constants containing C. If an ordering of a set of constants containing C is such that the first k elements of that ordering are $c_1,...,c_k$, we say that it is a C-ordering. Note that, by construction, $R_N$ represents a C-ordering.

Once $R_N$ has been constructed (steps 1-3), TL can be used to simulate "counter programs". Counter programs are programs using a fixed number of integer variables (which we denote i, j, k,...) which can be incremented and decremented by one, and tested for zero. The language also contains a "while" loop and an assignment. By [HU], any partial recursive function on integers can be computed using a counter program. Following is an example of a counter program which computes in k the product of i and j:

```
k := 0
while i ≠ 0 do
  l := j;
  while l ≠ 0 do k := k+1; l := l-1 done;
  i := i-1
  done.
```

Counter programs can be simulated by TL as follows. Each integer variable i is represented by a unary relation $R_i$. If the current value of i is j, then $R_i$ contains exactly the constant $a_j$ provided by relation $R_N$. Increments and decrements are simulated as follows:

*i := i+1:*
if $R_i(x)$ /\ $R_N(xy)$ then do
  $d_{R_i}(x)$;
  if y ≠ $ then do $i_{R_i}(y)$ done;
  if y = $ then do
    with new α do extend(α); $i_{R_i}(\alpha)$ done
    done
  done

*i := i-1:*
The simulation is similar (note that counter programs can be assumed not to attempt subtracting from zero).

Finally, a while loop with a test i = 0 is simulated by a while transaction with the condition $R_i(¢)$. Thus, TL can be used to simulate counter programs. We will therefore use counter programs in conjunction with TL programs whenever needed. We will also use arithmetic and boolean expressions computable using counter programs. It is understood that such counter programs denote TL programs simulating them.

We next show that TL can be used to compute (and decode) the Godel number of a database. Let **R** be a database schema and **I** an instance of **R**. In order to define the Godel number of **I**, we first define the Godel number g(u) of an n-tuple $<a_{i_1}...a_{i_n}>$ as $p_1^{i_1}...p_n^{i_n}$, where $p_i$ is the i-th prime (each constant $a_{i_j}$ denotes the integer $i_j$). The Godel number g(R) of an n-ary relation R with r tuples $\{u_1,...,u_r\}$ is $p_1^{g(u_1)}...p_r^{g(u_r)}$, where $p_i$ is the i-th prime and each $u_i$ is lexicographically less than $u_{i+1}$ (according to the order on constants induced by $R_N$). Finally, the Godel number of **I**, denoted g(**I**), is defined by $g(\mathbf{I}) = p_1^{g(I(R_1))}...p_q^{g(I(R_q))}$, where **R** = $\{R_1,...,R_q\}$, and $p_i$ is the i-th prime. Clearly, there is a one-to-one correspondence between each instance **I** of **R** and the Godel number g(**I**) (given a fixed $R_N$).

We now show that TL can compute the Gödel number of **I**, in a counter variable g(**I**) (represented in TL by a unary relation $R_{g(I)}$). We will use the following integer variables:

$p_i$ holds the i-th prime, necessary for computing g(u) for a tuple u ($1 \leq i \leq m$ where $m = \max(\{q\} \cup \{arity(R) \mid R \in \mathbf{R}\})$.

p holds consecutive primes (necessary for computing $g(\mathbf{I}(R))$).

g(u) holds the Godel number of a tuple.

$g(R_i)$ holds $g(\mathbf{I}(R_i))$ $(1 \leq i \leq q)$.

$g(\mathbf{I})$ will hold $g(\mathbf{I})$.

In addition, we will use q relations $R'_i$ $(1 \leq i \leq q)$, which will hold copies of the relations $R_i$.

The program also uses two procedures which we now describe separately. The first procedure is next(p), which, for a given value of the counter variable p, computes in p the next larger prime. Clearly, next(p) can be realized by a counter program. The second procedure, $\min(R'_i,Q'_i)$ takes as input a relation $R'_i$ of arity $m_i$ and produces in $Q'_i$ the lexicographically minimum tuple of $R'_i$. The procedure uses a relation $S'_i$ with arity $m_i$. Note that a test a>b can be evaluated using a counter program.

$\min(R'_i,Q'_i)$:
while $R'_i(u_1,\ldots,u_{m_i}) \wedge R'_i(v_1,\ldots,v_{m_i}) \wedge u_1 > v_1$ do
  $i_{S'_i}(u_1,\ldots,u_{m_i})$ done;
while $R'_i(u_1,\ldots,u_{m_i}) \wedge R'_i(v_1,\ldots,v_{m_i}) \wedge u_1 = v_1 \wedge u_2 > v_2$ do
  $i_{S'_i}(u_1,\ldots,u_{m_i})$ done;
. . .
while $R'_i(u_1,\ldots,u_{m_i}) \wedge R'_i(v_1,\ldots,v_{m_i}) \wedge u_1 = v_1 \wedge \ldots$
  $\wedge\ u_{m_i-1} = v_{m_i-1} \wedge u_{m_i} > v_{m_i}$ do
  $i_{S'_i}(u_1,\ldots,u_{m_i})$ done;
if $R'_i(u_1,\ldots,u_{m_i}) \wedge \neg S'_i(u_1,\ldots,u_{m_i})$ then do $i_{Q'_i}(u_1,\ldots,u_{m_i})$ done

The computation of $g(\mathbf{I})$ proceeds as follows:

1. initialize all counter variables to 1;
2. compute in each $p_i$ the i-th prime;
3. for each relation $R_i$ $(1 \leq i \leq q)$ of arity $m_i$ do

```
p := 2;          (* first prime *)
R'_i ← R_i;
while R'_i(u_1,...,u_{m_i}) do
  erase_{Q'_i}; min(R'_i,Q'_i);
  if Q'_i(v_1,...,v_{m_i}) then do
    g(u) := p_1^{v_1}...p_{m_i}^{v_{m_i}}; d_{R'_i}(v_1,...,v_{m_i})
    done
  g(R_i) := p^{g(u)}. g(R_i);
  next(p)
  done
g(I) := p_i^{g(R_i)}. g(I)
done.
```

It is easily verified that the above program always stops and that the final value of $g(\mathbf{I})$ is the Godel number of **I**. Conversely, given the Godel number $g(\mathbf{I})$ of a database, the database **I** can be reconstructed by a straightforward "reversal" of the encoding procedure described above.

We are ready to prove the main result of this section:

*Theorem 3.1.1:* TL is update complete.

*Proof:* Let $\rho$ be an r.e., C-generic relation on $\mathbf{inst}(\mathbf{R}) \times \mathbf{inst}(\mathbf{S})$, where **R** and **S** are database schemas. Let $C = \{c_1,...,c_k\}$. Consider the set

$G(\rho) = \{2^{g(I)} \cdot 3^{g(J)} \mid <\mathbf{I},\mathbf{J}> \in \rho,$ $g(\mathbf{I})$ and $g(\mathbf{J})$ are the Godel numbers of **I** and **J** with respect to an arbitrary C-ordering of $\mathbf{const}(\mathbf{I},\mathbf{J}) \cup C\}$.

Since $\rho$ is r.e., the set $G(\rho)$ is r.e. Thus, there exists a total recursive function $f_\rho$: $N \rightarrow N$ such that $\text{range}(f_\rho) = G(\rho)$. Let $P_\rho(i,g)$ be a counter program which, given some integer i, computes $f_\rho(i)$ in variable g. ($P_\rho(i,g)$ also denotes the corresponding TL program.)

We next describe a transaction t in TL, over the i-o schema $<\mathbf{R},\mathbf{S}>$, such that $\text{eff}_{<\mathbf{R},\mathbf{S}>}^{(+)} = \rho$. The transaction t consists of the following steps.

1) Compute in $R_N$ a C-ordering of all constants in $\mathbf{const}(\mathbf{I}) \cup C$.

2) Compute in the counter variable i the Godel number of **I** with respect to the ordering in $R_N$.

3) Non-deterministically find all instances **J** of **S** such that $2^{g(I)} \cdot 3^{g(J)} \in G(\rho)$, where $g(I)$ and $g(J)$ are the Godel numbers of **I** and **J** with respect to the ordering defined by $R_N$.

Note that $R_N$ is extended in the course of the computation to at least the constants in **J**. Step (3) is outlined below in more detail (T is a unary relation used to simulate the non-determinism).

```
i_T(0); i_T(1);
i := 0;
while T(x) do
  i := i+1;
  if x = 1 then do
    P_ρ(i,g);
    compute I' and K such that g = 2^g(I').3^K;
    if I = I' then do
      decode J from K = g(J);
      d_T(0); d_T(1);
      done
    done
  done
```

We show that $eff_{<\mathbf{R},\mathbf{S}>}(t) = \rho$, by double inclusion. Let $<\mathbf{I},\mathbf{J}> \in eff_{<\mathbf{R},\mathbf{S}>}(t)$. By construction, $<g(\mathbf{I}),g(\mathbf{J})> \in G(\rho)$ for some Godel numbering g w.r.t. some C-ordering of $\mathbf{const}(\mathbf{I},\mathbf{J})\cup C$. By the definition of $G(\rho)$, $<g(\mathbf{I}),g(\mathbf{J})> = <g'(\mathbf{I}'), g'(\mathbf{J}')>$ where $<\mathbf{I}',\mathbf{J}'> \in \rho$ and g' is computed with respect to some C-ordering of $\mathbf{const}(\mathbf{I}',\mathbf{J}')\cup C$. Then $<\mathbf{I},\mathbf{J}>$ is an isomorphic image of $<\mathbf{I}',\mathbf{J}'>$ under some isomorphism which is the identity on C. Since $\rho$ is C-generic and $<\mathbf{I}',\mathbf{J}'> \in \rho$, $<\mathbf{I},\mathbf{J}> \in \rho$.

Consider the reverse inclusion. Let $<\mathbf{I},\mathbf{J}> \in \rho$, and consider a C-ordering $\theta$ of $\mathbf{const}(\mathbf{I},\mathbf{J})\cup C$ such that the constants in C are followed by those in $\mathbf{const}(\mathbf{I})$ - C, which are followed by those in $\mathbf{const}(\mathbf{J})$-($\mathbf{const}(\mathbf{I})\cup C$). By definition, $<g(\mathbf{I}),g(\mathbf{J})> \in G(\rho)$, where $g(\mathbf{I})$ and $g(\mathbf{J})$ are the Godel numbers of **I** and **J** with respect to the ordering $\theta$. Clearly, step 1) leads to a C-ordering of $\mathbf{const}(\mathbf{I}) \cup C$. By the C-genericity of TL, one possible outcome of step (1) yields in $R_N$ an ordering equal to the restriction of $\theta$ to $\mathbf{const}(\mathbf{I})\cup C$. Also, one possible outcome of further extensions of $R_N$ consists of using the constants in $\mathbf{const}(\mathbf{J})$-($\mathbf{const}(\mathbf{I})\cup C$) following those in ($\mathbf{const}(\mathbf{I})\cup C$), in the order $\theta$. Therefore, one outcome for $R_N$ yields the ordering $\theta$ on the constants in $\mathbf{const}(\mathbf{I},\mathbf{J})\cup C$ (possibly followed by other constants).

Since $<g(\mathbf{I}),g(\mathbf{J})>$ is in $G(\rho)$, $<g(\mathbf{I}),g(\mathbf{J})> = f_\rho(i)$ for some integer i. Therefore, one possible outcome of step 3 is the instance obtained by decoding $g(\mathbf{J})$ with respect to the ordering in $\theta$, i.e., one possible outcome is $\mathbf{J}$. Thus, $<\mathbf{I},\mathbf{J}> \in \text{eff}_{<\mathbf{R},\mathbf{S}>}(t)$ and the proof is complete. □

*3.2 Deterministic update completeness of detTL*

In this section, we show that detTL is deterministic update complete. The proof is similar to that of the update completeness of TL. The main difference concerns the manner in which integers are represented by the values in the domain.

A difficulty arises from the fact that the determinism together with the C-genericity of detTL make it impossible, in general, to construct an ordering of values in the domain, as was done for TL using the relation $R_N$. The solution consists of constructing in $R_N$ *all* possible orderings of the relevant set of constants. For instance, the relation $R_N$ produced for the set of values {a,b} is represented in Figure 3.2.1. The values $\alpha$, $\beta$ are new values used to identify each ordering of the constants. Then, the computation proceeds by essentially carrying out the computation outlined for TL, in parallel, for each possible ordering of the relevant values. Due to C-genericity, the final result will be the same for all C-orderings.

Another difficulty of the construction arises from the fact that deletions cannot be used straightforwardly due to the semantics of detTL. Thus, logical deletions will sometimes be simulated in a roundabout fashion, using inserts (as in the proof of Theorem 2.4.1).

We now describe the main steps of the construction. By the definition of deterministic completeness, we have to show that, for each partially recursive, C-generic mapping $\rho$ from

| A | B | C |
|---|---|---|
| ¢ | a | $\alpha$ |
| a | b | $\alpha$ |
| b | $ | $\alpha$ |
| ¢ | b | $\beta$ |
| b | a | $\beta$ |
| a | $ | $\beta$ |

**Figure 3.2.1: $R_N$**

$\mathbf{inst(R)}$ to $\mathbf{inst(S)}$ there exists a transaction t in detTL such that $\mathrm{eff}_{<\mathbf{R},\mathbf{S}>}(t) = \rho$ (we identify the mapping $\rho$ with its graph).

As in the previous section, we can define the Godel number $g(\mathbf{I},\alpha)$ of an instance $\mathbf{I}$ with respect to a C-ordering $\alpha$ which includes the constants in $\mathbf{I}$. Let $\alpha_\infty$ be a fixed C-ordering of the entire domain. Since $\rho$ is computable, there exists a partial recursive function $f_\rho$ which maps the Godel number of $\mathbf{I}$ w.r.t. $\alpha_\infty$ to the Godel number of $\rho(\mathbf{I})$ w.r.t. $\alpha_\infty$, i.e.,

$$(+)\ f_\rho(g(\mathbf{I},\alpha_\infty)) = g(\rho(\mathbf{I}),\alpha_\infty).$$

Note that, due to the C-genericity of $\rho$, $f_\rho$ is independent of the ordering chosen. Indeed, let $\alpha$ be any C-ordering which includes the constants in $\mathbf{I}$. Clearly, there exists a one-to-one mapping h on $\mathbf{const(I)} \cup C$ such that $h|_C = \mathrm{id}$ and

$$(++)\ \text{for each } \mathbf{J} \text{ such that } \mathbf{const(J)} \subseteq \mathbf{const(I)} \cup C,$$
$$g(\mathbf{J},\alpha) = g(h(\mathbf{J}),\alpha_\infty).$$

(h maps the i-th element of $\mathbf{const(I)} \cup C$ according to $\alpha$ to the i-th one according to $\alpha_\infty$.) Then

$$\begin{aligned} f_\rho(g(\mathbf{I},\alpha)) &= f_\rho(g(h(\mathbf{I}),\alpha_\infty)) \text{ by } (++), \\ &= g(\rho(h(\mathbf{I})),\alpha_\infty) \text{ by } (+), \\ &= g(h(\rho(\mathbf{I})),\alpha_\infty) \text{ by C-genericity}, \\ &= g(\rho(\mathbf{I}),\alpha) \text{ by } (++). \end{aligned}$$

(Note that $\mathbf{const}(\rho(\mathbf{I})) \subseteq \mathbf{const(I)} \cup C$, since $\rho$ is a C-generic mapping.)

Now, let $P_\rho(g_1,g_2)$ be a counter program which, given a Godel number $g(\mathbf{I},\alpha)$ in $g_1$ produces $f_\rho(g(\mathbf{I},\alpha))$ in $g_2$. The construction now proceeds as follows:

1) Compute in $R_N$ all C-orderings of $\mathbf{const(I)} \cup C$

2) for each C-ordering $\alpha$ in $R_N$, compute (in parallel)
   (i) $g(\mathbf{I},\alpha)$ in $g_1$
   (ii) $f_\rho(g(\mathbf{I},\alpha)) = g(\rho(\mathbf{I}),\alpha)$ in $g_2$ using $P_\rho(g_1,g_2)$
   (iii) decode $\rho(\mathbf{I})$ from $g(\rho(\mathbf{I}),\alpha)$.

In accordance with the deterministic semantics of detTL, the result of step 2) will be the union of the $\rho(\mathbf{I})$ for each ordering $\alpha$. By the remarks on $f_\rho$ above, $\rho(\mathbf{I})$ computed at step 2 (iii) is independent of the ordering $\alpha$. So the final result (i.e. the union of the $\rho(\mathbf{I})$) equals each individual $\rho(\mathbf{I})$ computed for given $\alpha$.

We next discuss each of the above steps in more detail, starting with the construction of

$R_N$ and the representation of integers and counters. We first note that (deterministic) if statements and append operations can be simulated in detTL:

*If <condition> then do <statement> done:*
  $i_T(0)$;
  while <condition> /\ T(0) do
    <statement>; $d_T(0)$
    done

(The semantics of the above "if" statement is the obvious deterministic semantics analogous to that of the "while" statement.)

$R \overset{+}{\leftarrow} S$: (append S to R):
  if S(x) then do $i_R(x)$ done
    done

$R \overset{+}{\leftarrow} \Pi_A(S)$: (append to a unary relation R the projection of S on an attribute A)
  similar to above (details omitted).

Suppose now that the instance **I** of **R** and $C = \{c_1,...,c_k\}$ are given. Relation $R_N$ is constructed as follows (as before, we use "for each" as a meta-constructor ; T is a temporary, binary relation recording which domain elements have been included so far in each of the marked orderings):

for each $R_i$ in $\mathbf{R}$ and attribute A of $R_i$ do $D \overset{+}{\leftarrow} \Pi_A(R_i)$ done
with new $\alpha$ do
  $i_{R_N}(\text{¢ } c_1\ \alpha)$; $i_{R_N}(c_1\ c_2\ \alpha)$;...; $i_{R_N}(c_k\ \$\ \alpha)$;
  $i_T(c_1\alpha)$; $i_T(c_2\alpha)$;...$i_T(c_k\alpha)$
  done
while $R_N(x_1\ x_2\ \beta) \wedge \neg$ INCOMPLETE$(\beta) \wedge D(x) \wedge \neg T(x\beta)$ do
  with new $\gamma$ do
    if $R_N(y\ \$\ \beta)$ then do
      $i_{R_N}(y\ x\ \gamma)$; $i_{R_N}(x\ \$\ \gamma)$
      done;
    if $R_N(zv\beta) \wedge v \neq \$$ then do
      $i_{R_N}(zv\gamma)$
      done;
    if $R_N(uw\gamma)$ then do $i_T(u\gamma)$ done
    done;
    $i_{\text{INCOMPLETE}}(\beta)$
  done

| A | B | C |
|---|---|---|
| . | . | . |
| . | . | . |
| . | . | . |
| ¢ | $c_1$ | $\alpha_\sigma$ |
| $c_1$ | $c_2$ | $\alpha_\sigma$ |
| ... | ... | ... |
| $c_k$ | $a_{\sigma(1)}$ | $\alpha_\sigma$ |
| $a_{\sigma(1)}$ | $a_{\sigma(2)}$ | $\alpha_\sigma$ |
| ... | ... | ... |
| $a_{\sigma(n)}$ | $ | $\alpha_\sigma$ |
| . | . | . |
| . | . | . |
| . | . | . |

**Figure 3.2.2: $R_N$**

The construction of $R_N$ involves the consecutive construction of orderings for increasingly large subsets of const(I)∪C. Eventually, orderings for all constants in const(I)∪C are obtained. However, incomplete orderings obtained in the course of the computation cannot be straightforwardly deleted, due to the semantics of detTL. Instead, a record is kept of the

incomplete orderings in the unary relation INCOMPLETE. The orderings $\alpha$ used in Step 2 of the main construction outlined earlier are those not occurring in INCOMPLETE.

Let $\Sigma_n$ be the set of permutations of {1,...,n}. It is easily verified that the relation $R_N$ constructed by the above program from a given $C = \{c_1,...,c_k\}$ and additional constants $\{a_1,...,a_n\}$ in **I**, consists of the union for all $\sigma \in \Sigma_n$, of the relations represented in Figure 3.2.2. (The orderings in INCOMPLETE are omitted). Here $\alpha_\sigma$ is a domain value identifying the ordering corresponding to each $\sigma$. Note that each $c_i$ represents integer i in all orderings, and $a_{\sigma(j)}$ represents integer k+j in each ordering $\alpha_\sigma$. Thus, the relation $R_N$ obtained in the above program represents k+n integers. If additional integers are needed for an ordering $\alpha$, $R_N$ can be extended by one with respect to $\alpha$, using a new value a, as follows:

*extend(a,α)*:
  if $R_N$(x $ α) then do
    $d_{R_N}$(x $ α);
    $i_{R_N}$(x a α); $i_{R_N}$(a $ α)
    done

Given $R_N$ constructed as above, let c(i,α) be the constant corresponding to integer i with respect to the ordering $\alpha$. In step 2) above, in the context of an ordering $\alpha$, a counter variable j containing integer i can be represented as a unary relation $R_j$ containing the tuple <c(i,α)>. Increments and decrements (with respect to $\alpha$) are simulated as follows:

*j := j+1 (α)*:
  if $R_j$(x) then do
    if $R_N$(x $ α) then do
      with new z do extend(z,α) done
      done
    if $R_N$(x y α) then do
      $d_{R_j}$(x); $i_{R_j}$(y)
      done
    done.

*j := j-1 (α)*: The simulation is similar to that for increment and is omitted.

Given the above, it is now clear that detTL can simulate counter programs, with respect to each of the orderings in $R_N$.

Clearly, the encoding and decoding of a given database instance into its corresponding Godel number with respect to each ordering $\alpha$ can be done in a manner similar to TL. Thus, we have shown that the construction outlined earlier in the section (steps 1-2) can be implemented using detTL. It then follows:

*Theorem 3.2.1:* detTL is deterministic update complete. □

*3.3 The power of restricted languages.*

In this section, we characterize the classes of updates computable using the languages WTL, STL, and SdetTL.

The characterization for WTL follows easily from the completeness of TL. Informally, WTL can compute updates with a "finite amount" of non-determinism, that is, with a finite number of possible outcomes for each input instance. Due to C-genericity, this is equivalent to allowing in the result of the update of an instance **I**, only constants from **I** and C. Indeed, we have:

*Theorem 3.3.1:* WTL is finitely non-deterministic complete.

*Proof:* By the definition of finitely non-deterministic completeness, we have to show:

(*) For each i-o schema $<\mathbf{R},\mathbf{S}>$, $\{\text{eff}_{<\mathbf{R},\mathbf{S}>}(t) \mid t \in \text{WTL}\} = \{\rho \mid \rho$ is a C-generic, r.e. relation over $\text{inst}(\mathbf{R}) \times \text{inst}(\mathbf{S})$ such that $(\mathbf{I},\mathbf{J}) \in \rho$ implies that $\text{const}(\mathbf{J}) \subseteq \text{const}(\mathbf{I}) \cup C\}$.

Consider a transaction t in WTL, over the i-o schema $<\mathbf{R},\mathbf{S}>$. Due to the weak safety, if $(\mathbf{I},\mathbf{J}) \in \text{eff}_{<\mathbf{R},\mathbf{S}>}(t)$ then $\text{const}(\mathbf{J}) \subseteq \text{const}(\mathbf{I}) \cup C$, whence the first inclusion. Conversely, let $\rho$ be a C-generic, r.e. relation over $\text{inst}(\mathbf{R}) \times \text{inst}(\mathbf{S})$ such that $(\mathbf{I},\mathbf{J}) \in \rho$ implies $\text{const}(\mathbf{J}) \subseteq \text{const}(\mathbf{I}) \cup C$. By Theorem 3.1.1, there is a transaction t such that $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t) = \rho$. By definition, t is safe w.r.t. $<\mathbf{R},\mathbf{S}>$. By Theorem 2.3.1, there is a transaction t' in WTL which is $<\mathbf{R},\mathbf{S}>$-equivalent to t. Thus $\rho = \text{eff}_{<\mathbf{R},\mathbf{S}>}(t')$ for some t' in WTL. □

We next consider the power of STL. We will argue that STL computes exactly the database updates in NPSPACE. We start by defining the complexity classes NPSPACE and PSPACE for updates.

*Notation:* For each instance **I** over a database schema **R**, tape(**I**) denotes a standard encoding[2] of **I** on a Turing Machine (TM) tape.

*Definition:* A database update $\rho$ is in NPSPACE (respectively, PSPACE) iff there exists a non-deterministic (respectively, deterministic) TM M such that:

(i) under input tape(**I**), the set of tapes reached by M in an accepting state is {tape(**J**) | (**I**,**J**) $\in \rho$}, and

(ii) there exists a polynomial P such that M does not construct a tape larger than P(|tape(**I**)|) under input tape(**I**).

Since so far we simulated counter programs rather than TM's using TL, we will use an equivalent definition of NPSPACE updates in terms of non-deterministic counter programs (that is, counter programs which may allow a non-deterministic choice of the next instruction, from a finite set of instructions). Note that STL can simulate the non-determinism in counter programs. Indeed, if at some point in the counter program, the next instruction is non-deterministically chosen from a set $\{s_1,...,s_k\}$ of instructions, this can be simulated as follows (T is a unary temporary relation):

$i_T(1);...i_T(k);$
if T(x) then do
  if x=1 then do $s_1$ done;
  ...
  if x=k then do $s_k$ done
  done.

We now present a characterization of NPSPACE in terms of counter programs. Intuitively, NPSPACE updates are updates computable using non-deterministic counter programs with exponentially bounded counters (since exponentially bounded counters can be simulated using polynomially bounded tapes, and conversely).

[2] Such an encoding is exhibited in the proof of Theorem 3.3.3.

*Lemma 3.3.2:* A database update $\rho$ from $\mathbf{inst(R)}$ to $\mathbf{inst(S)}$ is in NPSPACE iff there exist Godelizations $g_R$: $\mathbf{inst(R)} \rightarrow N$ and $g_S$ : $\mathbf{inst(S)} \rightarrow N$ and a non-deterministic counter program $P_\rho(i,j)$ such that

(i) $g_R$ and $g_S$ are exponentially bounded, that is, there exists a polynomial Q such that for each **I** and **J**, $g_R(\mathbf{I}) \leq 2^{Q(\#\mathbf{const(I)})}$ and $g_S(\mathbf{J}) \leq 2^{Q(\#\mathbf{const(J)})}$.

(ii) given $g_R(\mathbf{I})$ in i, $P_\rho(i,j)$ outputs non-deterministically in j exactly all $g_S(\mathbf{J})$ such that $(\mathbf{I},\mathbf{J}) \in \rho$, and there exists a polynomial P such that no counter variable of $P_\rho$ contains an integer larger than $2^{P(\#\mathbf{const(I)})}$ in the course of the computation.

*Proof:* Consider the "only if" part. Let $\rho$ be a database update from $\mathbf{inst(R)}$ to $\mathbf{inst(S)}$, in NPSPACE. Let M be a TM computing $\rho$ in NPSPACE. Suppose the tape alphabet of M is $\{a_1,...,a_k\}$ (we assume that the head and state are encoded in the tape symbols). For each tape $a_{i_0} \ldots a_{i_n}$ let $\mathrm{int}(a_{i_0}\ldots a_{i_n})$ be the integer

$$i_0 + (k+1)i_1+...+(k+1)^n i_n.$$

Note that int is a one-to-one function which is exponential in the length of the tape. Let $g_R(\mathbf{I})$ $(g_S(\mathbf{J}))$ be $\mathrm{int}(\mathrm{tape}(\mathbf{I}))$ $(\mathrm{int}(\mathrm{tape}(\mathbf{J})))$. Clearly, $g_R(\mathbf{I})$ $(g_S(\mathbf{J}))$ is exponential in $\#\mathbf{const(I)}$ $(\#\mathbf{const(J)})$. Let $P_\rho(i, j)$ be a non-deterministic counter program which, under input $g_R(\mathbf{I}) = \mathrm{int}(\mathrm{tape}(\mathbf{I}))$, simulates in the obvious fashion the computation of M on $\mathrm{tape}(\mathbf{I})$, and outputs $g_S(\mathbf{J}) = \mathrm{int}(\mathrm{tape}(\mathbf{J}))$ in j whenever M reaches an accepting state. Since M constructs, during the computation, tapes of length bounded by a polynomial in $|\mathrm{tape}(\mathbf{I})|$, it is clear that $P_\rho$ can simulate the computation of M using counter variables bounded by $2^{P(\#\mathbf{const(I)})}$ for some polynomial P. This completes the "only if" part of the proof. The "if" part is similar, and is based on the fact that an exponentially bounded counter can be simulated by a TM using a polynomially bounded tape. The details are straightforward, and are omitted. □

It is clear that an analogous result holds for PSPACE updates.

We now characterize the power of STL updates.

*Theorem 3.3.3:* The class of database updates computable by transactions in STL is the class of NPSPACE database updates.

*Proof:* Let t be a transaction in STL, over an i-o schema $<\mathbf{R},\mathbf{S}>$. Consider an instance **I** over

**R**. Since t is in STL, only constants in const(I)∪C are used in the computation of t under input **I**, where C is a finite set of constants. Thus, each database produced in the course of the computation of t on **I** contains a number of tuples polynomially bounded in #const(**I**). It then follows easily that t can be simulated by a TM using a tape polynomially bounded in |tape(**I**)|. Thus, each transaction in STL defines a database update in NPSPACE.

Conversely, suppose that $\rho$ is a database update in NPSPACE, from inst(**R**) to inst(**S**). First note that, from the C-genericity of $\rho$ and the fact that $\rho$ is in NPSPACE, it follows that, for each $(\mathbf{I},\mathbf{J}) \in \rho$, const(**J**) $\subseteq$ const(**I**) $\cup$ C. Indeed, if some value $\alpha$ not in const(**I**) $\cup$ C occurs in **J**, then, due to the C-genericity of $\rho$, there are infinitely many isomorphic **J** such that $(\mathbf{I},\mathbf{J}) \in \rho$. Given the fact that each output **J** must be representable on a tape polynomially-bounded in the input, this is a contradiction.

We now define the Godelization $g_R$ of instances of a given database schema **R**. The Godelization will be exponential in #const(**I**), and thus will be more economical than the Godelization described earlier which was based on factorization into primes. (That more standard Godelization was used for simplicity.) As before, $g_R$ is defined with respect to a given C-ordering of const(**I**) $\cup$ C, provided by a binary relation $R_N$ which is constructed non-deterministically. For each i, $1 \leq i \leq \#(\text{const}(\mathbf{I}) \cup C)$, let $a_i$ be the constant in const(**I**) $\cup$ C, corresponding to i (according to $R_N$). For the Godelization, we identify relations in **R** by an integer; one additional integer is used, intuitively, as a tuple separator. Let n = #(const(**I**)∪C). If $\mathbf{R} = \{R_1,...,R_p\}$, let $R_i$ be denoted by the integer n+i, and n+p+1 denote a tuple separator (say, >). Let $k_i$ be the arity of $R_i$. Consider the sequence (tape)

$$R_1\ a^1_{11}...\ a^1_{1k_1} > a^2_{11}...\ a^2_{1k_1} ... > R_2 ... > R_p\ a^1_{p1}...\ a^1_{pk_p} > ...$$

where each $R_i$ is followed by the tuples in $\mathbf{I}(R_i)$, separated by ">", in lexicographical order with respect to the pointwise ordering defined by $R_N$. Note that the length of the sequence is a polynomial P in n, and that there are n+p+1 distinct symbols in the sequence. Thus, the sequence can be uniquely represented by an integer of the form $\sum_{i=1}^{P(n)} x_i(n+p+2)^i$ (where $x_i$ is the integer between 1 and n+p+1 corresponding to the i-th symbol in the sequence), which is exponential in n. We define $g_R(\mathbf{I})$ as the above integer associated with **I**. The mapping $g_S$ is defined similarly. By Lemma 3.3.2, there exists a counter program $P_\rho(i,j)$ which, given $g_R(\mathbf{I})$ in i, computes in j $g_S(\mathbf{J})$, where $(\mathbf{I},\mathbf{J}) \in \rho$, and its counter variables are exponentially bounded in

n.

Clearly, $P_\rho$, $g_R$, and $g_S$ can be simulated in STL if counters exponentially bounded in the number of constants in the input database can be simulated. The simulation differs from the earlier simulation of counters in the representation of integers. We next show how to represent, using n constants, integers up to $2^{n^k}$ for some fixed k>0. Let $R_N$ be a binary relation representing, as before, a C-ordering of the n constants. Let $t_1,\ldots,t_{n^k}$ be the $n^k$ k-tuples with values in the n constants, ordered lexicographically by $R_N$. Let $a_0$, $a_1$ be two of the constants denoted by 0 and 1, respectively. Let $\mathbf{R}_N^k$ be a relation of arity k+1 where the first k columns contain the cartesian product of the n constants and the (k+1)-th column contains values in {0,1}. The relation $\mathbf{R}_N^k$ represents the integer whose binary representation is given by the values in column C, i.e., $\sum_{<t_i,1>\in \mathbf{R}_N^k} 2^{i-1}$. For instance, given $R_N$ represented in Figure 3.3.1 (a), $\mathbf{R}_N^2$ represented in Figure 3.3.1(b) denotes the integer 3.

Each counter i is represented in STL by a (k+1)-ary relation $R_i$ similar to the $\mathbf{R}_N^k$ described above. Clearly, increments and decrements of counter variables can be simulated by STL. Thus, $P_\rho$ can be simulated by STL. The computation of $g_R(\mathbf{I})$ and $g_S^{-1}(\mathbf{J})$ by STL is similar to that for the earlier Godelization, except that the correspondence between integers and database values is less direct, since an integer encoding a domain value is no longer denoted by the value itself. For instance, if $\mathbf{R}_N^2$ in Figure 3.3.1 represents the Godel number of a constant, the constant is the third one according to $R_N$, i.e., c. It is now straightforward to

| $R_N$ | A | B |
|---|---|---|
| | ¢ | a |
| | a | b |
| | b | c |
| | c | $ |

(a)

| $R_N^2$ | A | B | C |
|---|---|---|---|
| | a | a | 1 |
| | a | b | 1 |
| | a | c | 0 |
| | b | a | 0 |
| | b | b | 0 |
| | b | c | 0 |
| | c | a | 0 |
| | c | b | 0 |
| | c | c | 0 |

(b)

Figure 3.3.1

construct a transaction t in STL whose effect is $\rho$. Thus, every database update in NPSPACE is the effect of some transaction in STL. □

We finally consider the computational power of the language SdetTL. We show that the set of updates computable using SdetTL transactions without constants corresponds to the language *LE* of [CH2]. As a consequence,

- each query obtained with relational algebra extended with a fixpoint operation is computable by an SdetTL transaction,
- The set of updates computable by SdetTL is included in the set of PSPACE updates, and
- SdetTL cannot express queries such as "do $R_1$ and $R_2$ have equal cardinality?" [CH3]. Thus the set of updates computable by SdetTL transactions is strictly included in the set of PSPACE updates.

We now review briefly the language *LE*. The *LE* language has ranked variables. The value of a variable of rank i is a relation of arity i. The basic *LE* statement is of the form:

$$X \leftarrow t$$

where t is a relational algebra expression using input relations and relation variables, and X is a relation variable. Basic statements can be combined as follows: if $q_1$ and $q_2$ are statements, and X is a relation variable,

$(q_1; q_2)$, and

while $X \neq \{\}$ do $q_1$

are statements. A *query* q from **R** to **S** is a query such that the input relations are in **R**, and **S** is included in the set of relation variables used by q.

We call SdetTL$^-$ the set of SdetTL (p-)transactions without constants. Now we have:

*Theorem 3.3.4:* Let $<\mathbf{R}, \mathbf{S}>$ be an i-o schema with $\mathbf{R} \cap \mathbf{S} = \phi$. A mapping from inst(**R**) to inst(**S**) is computable by SdetTL$^-$ iff it is computable by *LE*.

*Proof (Sketch):* (if) Each relational algebra query can be simulated in SdetTL using temporary relations for storing results of subqueries. Temporary relations are also used for relation variables. A statement

$$\text{while } X \neq \{\} \text{ do } S$$

is simulated by

$$\text{while } X(x) \text{ do } t_S \text{ done}$$

where $t_S$ is the transaction corresponding to S. Thus, *LE* can be simulated by SdetTL$^-$.

(only if) let t be a transaction in SdetTL$^-$ over an i-o schema $<\mathbf{R},\mathbf{S}>$. Let $\mathbf{T}$ be the set of relations used by t. For each variable x occurring in t, let $A_x$ be a new attribute. Let V be the set of all such attributes. For each $X \subseteq V$, and each T in $\mathbf{T}$, we consider the relational schema $T \cup X$ that we denote by ext(T,X). The database schema ext($\mathbf{T}$,X) is obtained by extending with X each relation in $\mathbf{T}$. We also use an *LE* program $\text{reduce}_X$ which assigns to relations in $\mathbf{T}$ the relations obtained by projecting out the X-columns of the corresponding relation in ext($\mathbf{T}$,X).

We show that for each p-transaction s occurring in t, and each X such that $\{A_y \mid y \in \text{free}(s)\} \subseteq X \subseteq V$, there exists a query q(s,X) in *LE* which simulates s using relations in ext($\mathbf{T}$,X). Intuitively, the attributes in X hold parameter values; a tuple $u \times \mu$ is in ext(T,X) where u is over T and $\mu$ over X, iff the tuple u is in T when s is called with parameters given by $\mu$.

Formally, we prove inductively that:

(*) Let s be a transaction occurring in t, $W = \{x_1,\ldots,x_k\}$ with $\text{free}(s) \subseteq W \subseteq \text{var}(t)$, and $\hat{W} = \{A_{x_1},\ldots,A_{x_k}\}$. Then there exists a query $q(s,\hat{W})$ in *LE* such that for each $\mathbf{I}$ over ext($\mathbf{T}$,$\hat{W}$),

$$q(s,\hat{W})(\mathbf{I}) = \cup \{\mathbf{I}_\mu \mid \mu \text{ a tuple over } \hat{W}\}$$

where $\mathbf{I}_\mu$ is defined as follows: for each T in $\mathbf{T}$,

- $\mathbf{I}_\mu(\text{ext}(T,\hat{W})) = \mu \times (\mu s)(\text{reduce}_{\hat{W}}(\mathbf{J}_\mu))$;
- $(\mu s)$ is the transaction obtained by evaluating in s, for each k, $x_k$ by $\mu(A_{x_k})$ (recall that $\text{free}(s) \subseteq W$); and
- $\mathbf{J}_\mu$ is defined by: for each T in $\mathbf{T}$,
  $\mathbf{J}_\mu(\text{ext}(T,\hat{W})) = \sigma_{A_{x_1}=\mu(A_{x_1})/\backslash\ldots/\backslash A_{x_m}=\mu(A_{x_m})}(\ \mathbf{I}(\text{ext}(T,\hat{W}))\ )$.

Note that (*) suffices to conclude the proof. Indeed, suppose that (*) holds. For s = t and $\hat{W} = \phi$, (*) becomes $q(t,\phi)(\bar{\mathbf{I}}) = t(\bar{\mathbf{I}})$ where $\bar{\mathbf{I}}$ is an instance over $\mathbf{T}$, which shows that *LE* can simulate SdetTL$^-$.

As mentioned above, we prove (*) by induction. Four cases have to be considered:

(a) $s = i_T\ (x_1,...,x_i)$ ; then $q(s,\hat{W})$ is the assignement statement

$$ext(T,\hat{W}) \leftarrow ext(T,\hat{W}) \cup \sigma_{A_1=A_{x_1}/\backslash.../\backslash A_i=A_{x_i}}\ (Dom^{k+\#(\hat{W})})$$

where $T = \{A_1,...,A_i\}$.

(b) deletions are treated similarly (with difference instead of union).

(c) $s = s_1;s_2$; let W be such that $free(s) \subseteq W \subseteq var(t)$. Then for each i, $free(s_i) \subseteq W \subseteq var(t)$. Thus $q(s_1,\hat{W})$; $q(s_2,\hat{W})$ can be used.

(d) s = while cond do s' done ;
Let W and $\hat{W}$ be as in the statement of (*). Let $Z = var(cond) \cup W$. (Clearly, $free(s') \subseteq Z \subseteq var(t)$.) By induction, (*) holds for s', Z and $\hat{Z}$. We use $q(s',\hat{Z})$ to construct $q(s,\hat{W})$ as follows:

```
Q ← ψ_cond;
while Q ≠ φ do
  for each T in ext(T, Ẑ) do
    ext(T, Ẑ) ← ext(T,Ŵ) join Q;
    done
  q(s', Ẑ);
  reduce_{Ẑ−Ŵ};
  Q ← ψ_cond;
  done
```

Here $\psi_{cond}$ is a relational algebra expression with target $\hat{Z}$ corresponding to the condition of the loop (extended to $ext(\mathbf{T},\hat{Z})$). Note that a tuple $\mu$ over $\hat{Z}$ induces a valuation of var(cond). A tuple $\mu$ over $\hat{Z}$ in Q indicates that the instance over **T** corresponding to $\mu_{\hat{W}}$ satisfies $\mu$(cond).

In the previous program, the deterministic semantics of the while, based on the *union* of the outcomes for all valuations, is realized by the reduce operation which projects out the columns corresponding to the valuations. □

We have seen that the computational power of SdetTL is strictly less than the PSPACE updates. Next, we show that SdetTL can compute exactly the PSPACE updates, under the assumption that each database instance provides an ordering of all constants occurring in the instance. We call such a database instance *ordered*. Specifically, each ordered database contains a binary relation $R_N$, such as the one used in the previous section to simulate order, and all constants in the database occur in $R_N$. We now have:

*Theorem 3.3.4:* Let $<\mathbf{R}, \mathbf{S}>$ be an i-o schema such that $R_N \in \mathbf{R}$.

(i) For each transaction t in SdetTL, over the i-o schema $<\mathbf{R}, \mathbf{S}>$, $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t)$ is a deterministic update in PSPACE, and

(ii) for each deterministic update $\tau$ from $\mathbf{R}$ to $\mathbf{S}$ in PSPACE there exists a transaction t in SdetTL such that $\text{eff}_{<\mathbf{R},\mathbf{S}>}(t)|_\Delta = \tau|_\Delta$, where $\Delta$ is the family of all instances over $\mathbf{R}$ ordered by $R_N$.

*Proof:* (Sketch) Clearly, (i) is an immediate consequence of Theorem 3.3.3. Consider (ii). As mentioned above, an analog of Lemma 3.3.2 holds for PSPACE updates. Thus, the problem is reduced to simulating counters exponentially bounded by the number of constants in the input database. The simulation parallels the one for STL (Theorem 3.3.3), with the difference that the computation is deterministic, due to the fact that an ordering of all constants is provided in $R_N$. □

*Remark:* The notion of ordered database instance requires that an ordering of constants be explicitely provided in the database instance. One might consider more general notions of ordered instance. For example, one might require that an ordering of the constants be *computable* from the instance by a C-generic, deterministic computation. (This is equivalent to the requirement that the only automorphism of the instance be the identity mapping.) However, we do not pursue this issue further in the present paper.

The main results of Section 3 are summarized in Figure 3.3.2.

| Language | Expressive Power |
|---|---|
| TL | non-deterministic update complete |
| WTL | finitely non-deterministic update complete |
| STL | NPSPACE |
| detTL | deterministic update complete |
| SdetTL | FP $\subset$ SdetTL $\subset$ PSPACE |
| SdetTL on ordered instances | PSPACE |

**Figure 3.3.2**

## 4. SPECIFICATION COMPLETENESS

In the previous section, we characterized the updating power of our transaction languages. We now consider the specification power of these languages.

We first recall the concept of "transactional schema", introduced in [AV4], which provides both static and dynamic specifications associated with a transaction language. Consider, for instance, the TL language. Let **R** be a database schema. A *transactional schema* over **R** is a pair <**R**,T> (or T when **R** is understood) where T is a finite set of parameterized transactions. We explicitly identify the variables $x_1,...,x_n$ in a parameterized transaction t which are viewed as parameters by calling the p-transaction $t(x_1,...,x_n)$. The parameters $\{x_1,...,x_n\}$ of a parameterized transaction $t(x_1,...,x_n)$ must include the free variables of t. Let $t(x_1,...,x_n)$ be a parameterized transaction. A *transaction call* to t is a transaction vt where v is a valuation of $x_1,...,x_n$. In other words, a transaction call to t is obtained by substituting in t each parameter $x_i$ by the constant $v(x_i)$. Intuitively, given a transactional schema T, a legal

transaction is a call to some parameterized transaction in T. Formally, the transition set and database family defined by T are given by:

$$\text{tran(T)} = \cup \{ \text{eff}_{\text{TL}}(s) \mid s \text{ a call to some t in T} \}, \text{ and}$$
$$\text{gen(T)} = \{ \mathbf{I} \mid (\phi, \mathbf{I}) \in (\text{tran(T)})^* \}.$$

Transactional schema languages are defined for the WTL, STL, detTL and SdetTL in similar ways. The following example illustrates the use of TL to obtain transactional schemas.

*Example 4.1:* Consider a database consisting of two relations, TA(NAME, COURSE), and PHD(NAME, ADDRESS). It is assumed that each PhD student has exactly one address, each TA teaches one course, and each TA is also a PhD student. This is formalized by the following dependencies:

G= {NAME → COURSE, NAME → ADDRESS, TA(NAME) ⊆ PHD(NAME)}

Thus, the set of valid database instances can be statically described as the set of all instances satisfying G.

Now, consider the following two parameterized transactions in TL:

HIRE(x,y,z)=

while TA(x,v) do $d_{TA}(x,v)$ done; $i_{TA}(x,y)$;

while PHD(x,w) do $d_{PHD}(x,w)$ done; $i_{PHD}(x,z)$,

and

FIRE(x)= while TA(x,v) do $d_{TA}(x,v)$ done.

It is easily seen that gen({HIRE,FIRE}) = sat(G). Additionally, note that tran({HIRE,FIRE}) enforces the dynamic constraints: "once a PhD student, always a PhD student", and "all PhD students have been TA's at some point". Clearly, one can find parameterized transactions which would define the same set of valid states but enforce different dynamic constraints.

Since updating power is related to specification power, some of our results follow straightforwardly from results already established on updating power. Indeed, from Theorems 3.1.1 and Proposition 1.3.2, we have :

*Fact:* TL is (dynamic) specification complete.

The next result concerns the specification power of WTL. Intuitively, the relevant

difference between TL and WTL with respect to specification is that there is a bound on the number of constants that transactions in WTL can add to any given instance. Indeed, we have.

*Theorem 4.1:* WTL is (dynamic) bounded specification complete.

*Proof:* Let $<\mathbf{R},T>$ be a transactional schema where each parameterized transaction in T is in the language WTL. Clearly, tran($<\mathbf{R},T>$) is a bounded transition set. Conversely, let $\rho$ be a bounded transition set over a database schema $\mathbf{R}$. In particular, $\rho$ is r.e. and C-generic for some finite C. Let $\mathbf{R}^{\infty}_{N}$ be an arbitrary C-ordering of the entire domain, and $g_{\mathbf{R}^{\infty}_{N}} : \rho \rightarrow \mathbf{N}$ a Godelization of $\rho$ with respect to $\mathbf{R}^{\infty}_{N}$, such as the ones considered in the previous section. Then $g_{\mathbf{R}^{\infty}_{N}}(\rho)$ is r.e., so there exists a partially recursive function f whose domain is $g_{\mathbf{R}^{\infty}_{N}}(\rho)$.

Note that, due to the C-genericity of $\rho$, f is independent of the ordering $\mathbf{R}^{\infty}_{N}$.

Let $P_f(i)$ be a counter program computing f(i). For each $\mathbf{I}$ and $\mathbf{J}$ over $\mathbf{R}$, let $C_{\mathbf{I},\mathbf{J}} =$ const($\mathbf{J}$) - const($\mathbf{I}$). Since $\rho$ is bounded, there exists an integer K such that for each $(\mathbf{I},\mathbf{J}) \in \rho$, $\#(C_{\mathbf{I},\mathbf{J}}) \leq K$. Let $t(x_1,...,x_K)$ be the parameterized transaction in WTL, outlined below ($\mathbf{I}$ is the input instance) :

1. compute, in a binary relation $R_N$, an arbitrary $(C \cup \{x_1,...,x_k\})$-ordering of const($\mathbf{I}$) $\cup$ C $\cup \{x_1,...,x_K\}$.
2. Non-deterministically generate an (output) instance $\mathbf{J}$ using constants in $C \cup \{x_1,...,x_K\} \cup$ const($\mathbf{I}$).
3. Compute in i, g(($\mathbf{I},\mathbf{J}$)) with respect to the current ordering $R_N$.
4. Simulate $P_f(i)$. (If $P_f$ terminates, then $\mathbf{J}$ is a resulting instance.)

The simulation of integers (in $R_N$) and counter variables is done as in the construction used to prove the update completeness of TL in Section 3.

We claim that tran($\{t(x_1,...,x_K)\}$) = $\rho$. First note that

tran($\{t(x_1,...,x_K)\}$) = $\cup$ {eff(t(v($x_1$),...,v($x_K$))) | v is a valuation of $\{x_1,...,x_K\}$}.

Let $(\mathbf{I},\mathbf{J}) \in$ tran($\{t(x_1,...,x_K)\}$). Then there exists a valuation v of $\{x_1,...,x_K\}$ such that $(\mathbf{I},\mathbf{J}) \in$ eff(t(v($x_1$),...v($x_K$))). From the construction of t, it follows that $P_f(g((\mathbf{I},\mathbf{J})))$ halts, so $(\mathbf{I},\mathbf{J}) \in \rho$. Conversely, let $(\mathbf{I},\mathbf{J}) \in \rho$. Let v be a valuation of $\{x_1,...,x_K\}$ such that v($\{x_1,...,x_K\}$) = $C_{\mathbf{I},\mathbf{J}}$. Consider t(v($x_1$),...,v($x_K$))($\mathbf{I}$). Since const($\mathbf{J}$) $\subseteq$ const($\mathbf{I}$) $\cup$ C $\cup$ {v($x_1$,...,v($x_K$)}, $\mathbf{J}$ is

non-deterministically

generated in step 2.

Since $(\mathbf{I},\mathbf{J}) \in \rho$, $P_f(g((\mathbf{I},\mathbf{J})))$ halts. Thus, $t(v(x_1),\ldots, v(x_K))$ non-deterministically outputs $\mathbf{J}$ under input $\mathbf{I}$, so

$(\mathbf{I},\mathbf{J}) \in \text{eff}(t(v(x_1),\ldots,v(x_K))$ and $(\mathbf{I},\mathbf{J}) \in \text{tran}(\{t(x_1,\ldots,x_K)\})$. □

By Proposition 1.3.1 and Theorem 4.1, we have:

*Fact:* WTL is not (dynamic) specification complete.

We next consider the specification power of detTL. We will use the following straightforward technical result :

*Lemma 4.2:* Let $t(x_1,\ldots,x_n)$ be a p-transaction and C the set of constants occurring in it. For each valuation v of $\{x_i \mid 1 \leq i \leq n\}$ such that $v(\{x_i \mid 1 \leq i \leq n\}) \cap C = \phi$, and each one-to-one homomorphism f of the domain such that $f|_C = id$,

(i) $f(\text{eff}_{TL}(vt)) = \text{eff}_{TL}(f(v(t)))$, and

(ii) $f(\text{eff}_{detTL}(vt)) = \text{eff}_{detTL}(f(v(t)))$.

*Proof:* (Sketch) From the definition of the semantics of TL and detTL it is clear that

$(\mathbf{I},\mathbf{J}) \in \text{eff}(v(t))$ iff

$(f(\mathbf{I}),f(\mathbf{J})) \in \text{eff}(f(v(t)))$.

It follows immediately that $f(\text{eff}(v(t))) = \text{eff}\,(f(v(t)))$. □

We now consider the specification power of detTL. We prove that detTL is deterministic specification complete.

*Theorem 4.3:* detTL is deterministic specification complete.

*Proof:* Let $<\mathbf{R},T>$ be a transactional schema in detTL. We first show that tran(T) is a deterministic transition set. Note that we can assume, wlog, that T consists of a single p-transaction. Indeed, if $T = \{t_1,\ldots,t_n\}$ and $k = \max\{k_i \mid t_i \text{ has } k_i \text{ parameters}, 1 \leq i \leq n\}$, let

$t(x_0,...,x_k)$ be the following ("for each" is a meta-construct whose semantics was explained in the previous section) :

for each i, $1 \leq i \leq k$, do
  if $x_0=a_i$ then do $t_i(x_1,...,x_{k_i})$ done
  done
while $x_0 \neq a_1 \wedge ... \wedge x_0 \neq a_n$ do nothing done

where the $a_i$ are distinct domain values, $1 \leq i \leq n$, and nothing is a transaction which does not modify the state of the database.

Clearly, tran({t}) = tran(T). Thus, assume T consists of a single transaction $t(x_0,...,x_k)$. Let $C = \{c_1,...,c_m\}$ be the set const(t), and let $a_0,...,a_k$ be distinct constants not in C. Let V be the set of valuations of $\{x_0,...,x_k\}$ into $C \cup \{a_0,...,a_k\}$, and let $\tau_v$=eff(vt), for each $v \in V$.

We claim that

(*) $\text{tran}(\{t\}) = \cup \{f(\tau_v) \mid v \in V, f \text{ is a domain isomorphism such that } f|_C = id\}$

Clearly, (*) establishes that tran({t}) is a deterministic transition set. To see (*), consider first $(\mathbf{I},\mathbf{J}) \in \text{tran}(\{t\})$. Then there exists a valuation w of $\{x_0,...,x_k\}$ such that $(\mathbf{I},\mathbf{J}) \in \text{eff}(wt)$. Clearly, there exists $v \in V$ such that, $w(x_i) = w(x_j)$ iff $v(x_i) = v(x_j)$ $(i \neq j)$ and $w(x_i) = c_j$ iff $v(x_i) = c_j$. Let f be the isomorphism of the domain such that $f(v(x_i)) = w(x_i)$, $0 \leq i \leq k$, and f is the identity everywhere else (note that f is implicitly the identity on C). By Lemma 4.2, $\text{eff}(wt) = f(\text{eff}(vt)) = f(\tau_v)$. Thus, $(\mathbf{I},\mathbf{J}) \in \cup \{f(\tau_v) \mid v \in V, f \text{ is a domain isomorphism such that } f|_C = id\}$.

Conversely, let $(\mathbf{I},\mathbf{J}) \in f(\tau_v)$ for some $v \in V$ and domain isomorphism f such that $f|_C = id$. Now $f(\tau_v) = f(\text{eff}(vt)) = \text{eff}(f(v(t)))$, by Lemma 4.2. Hence, $(\mathbf{I},\mathbf{J}) \in \text{tran}(\{t\})$, and (*) is established. Thus, tran({t}) is a deterministic transition set for every t.

Consider now a deterministic transition set

$$H = \cup \{f(\tau_i) \mid 1 \leq i \leq n, f \text{ is an isomorphism, } f|_C = id\},$$

where C is a finite set of constants and each $\tau_i$ is a database mapping which is $(C_i \cup C)$-generic for some finite $C_i$ $(1 \leq i \leq n)$. To conclude the proof, it suffices to show that H is the transition set of a transactional schema in detTL. We can assume, wlog, that $C_i \cap C = \phi$. Let $C_i = \{c_{ij} \mid 1 \leq j \leq k_i\}$. We will construct a transactional schema $T = \{t_i(x_1,...,x_{k_i}) \mid 1 \leq i \leq n\}$ such that

tran(T) = H. By Theorem 3.2.1, for each $\tau_i$ there exists a transaction $t_i$ in detTL such that $eff(t_i) = \tau_i$. Let $\bar{t}_i(x_1,...,x_{k_i})$ be the p-transaction obtained from $t_i$ by replacing all occurrences in $t_i$ of each constant $c_{ij}$ in $C_i$ by a variable $x_j$, $1 \leq j \leq k_i$. We show that tran(T) = H. For each i,

$tran(\{\bar{t}_i\}) = \cup \{ eff(v\bar{t}_i) \mid v$ a valuation of $x_1,...,x_{k_i} \}$

$= \cup \{f(\tau_i) \mid f$ is an isomorphism, $f|_C = id\}$ by Lemma 4.2.

Thus

$$tran(T) = \bigcup_{i=1}^{n} tran(\{\bar{t}_i\})$$

$= \cup \{f(\tau_i) \mid 1 \leq i \leq n$, f is an isomorphism, $f|_C = id\}$,

$= H$. □

We mentioned in the first section that there are bounded transition sets which are not deterministic. Thus, in particular, we have:

*Fact:* detTL is not bounded specification complete.

From Theorem 4.3, it follows that, for each transactional schema $<\mathbf{R},T>$ where T consists of parameterized transactions in detTL, gen(T) is a deterministic database family. Using this fact, we can now prove Theorem 1.4.2 (which is recalled here for convenience).

*Theorem 1.4.2:* For each finite set G of EGD's and total TGD's, sat(G) is a deterministic database family.

*Proof:* Let $\mathbf{R} = \{R_1,...,R_k\}$ be the database schema corresponding to G. We will exhibit a transactional schema $<\mathbf{R},T>$ over SdetTL, such that gen(T) = sat(G). This proves that sat(G) is a deterministic family. Let T consist of k parameterized transactions $t_i(u_i)$ defined as follows (P is a unary, temporary relation):

$t_i(u_i)$ ($u_i$ is a tuple of distinct variables over $R_i$):

$i_{R_i}(u_i)$; $i_P(0)$;
while P(0) do
  $d_P(0)$;
  for each total TGD $\varphi \Longrightarrow R(u)$ in G do
    while $\varphi \wedge \neg R(u)$ do $i_P(0)$; $i_R(u)$ done
    done

for each EGD $\varphi \Longrightarrow x{=}y$ in G do
  while $\varphi \wedge x \neq y$ do nothing done
  done
done

It is easily seen that, for each instance **I** over **R**, each i in [1..k], and each valuation v of $u_i$, if $t_i(v(u_i))$ halts on **I** then $t_i(v(u_i))(\mathbf{I}) \models G$. Thus, gen(T) $\subseteq$ sat(G). Conversely, if $\mathbf{I} \in$ sat(G), then **I** is obtained by the following set of calls to p-transactions in T, in some arbitrary order:

$$\{t_i(v(u_i)) \mid 1 \leq i \leq k, v(u_i) \in \mathbf{I}(R_i)\}.$$

Intuitively, each call $t_i(v(u_i))$ inserts in $R_i$ the tuple $v(u_i)$ and closes the resulting instance under the total TGD's of G. Since only tuples in **I** are inserted, and **I** satisfies the EGD's in G, no violation of the EGD's occurs, and all of the above calls terminate. It follows that sat(G) = gen(T). □

The specification powers of STL and SdetTL are not yet known. It is easy to see that they are not deterministic specification complete.

The results of the section are summarized in Figure 4.1.

| | |
|---|---|
| TL | specification complete |
| WTL | bounded specification complete |
| detTL | deterministic specification complete |

**Figure 4.1**

## CONCLUSIONS

In this paper, we proposed completeness criteria for transaction languages viewed as updating

and specification mechanisms. We then exhibited simple transaction languages which are complete with respect to various criteria proposed. The transaction languages use tuple insertion and deletion, a "while" loop, and random assignment of "invented" values. The five languages defined using those constructs differ along two lines:

(1) non-determinism vs. determinism, and

(2) unrestricted vs. "safe" use of invented values.

The results provided in this paper provide a fairly complete understanding of the connection between these features and the updating and specification power of corresponding languages. However, the specification aspect remains to be explored in more detail. With respect to dynamic specification, we have seen examples of transactional schemas whose transition sets satisfy certain dynamic constraints (e.g., "once a PhD student, always a PhD student"). It is of interest to investigate more formally the problem of dynamic constraints enforced by transactional schemas. In particular, this would require defining an appropriate language for expressing dynamic constraints. With respect to static specification, we established some connections between static specification by transactional schemas and specification using traditional static constraints. However, certain aspects of this connection require further investigation. One such aspect which appears to be of a fundamental nature is the connection between the complexity of checking satisfaction of static constraints defining a database family, and the complexity of generating the same family using a transactional schema.

Throughout our investigation, we repeatedly established connections between our update languages and existing query languages. In particular, we showed that our SdetTL has essentially the same computational power as the language LE defined in [CH2]. In general, we blurred the distinction between query languages and update languages. Intuitively, we called our languages "update" languages because they use single tuple inserts and deletes, which are traditionally viewed as update operations. However, the philosophical issue of when a language is an update or a query language, or whether such a distiction is indeed necessary, remains unresolved.

Recently, there has been a lot of interest in expressing queries and updates in a "declarative" fashion. In particular, stratified negation was introduced as a mechanism for increasing the power of Datalog programs. In a certain sense, stratification can be viewed as

Large Data Bases (1981).

[CCF] Castillo, I.M.V., M.A. Casanova, A.L. Furtado, A Temporal Framework For Data Bases, Proc., Int'l Conf. on Very Large Data Bases (1982).

[CFP] Casanova, M.A., R. Fagin, C.H. Papadimitriou, Inclusion Dependencies and their Interaction with Functional Dependencies, proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems (1981)

[CH1] Chandra, A.K., D. Harel, Computable Queries for Relational Databases, Journal of Computer and Systems Science (1980)

[CH2] Chandra, A.K., D. Harel, Programming Primitives for Database Languages, ACM Symposium on Principles of Programming Languages, (1981)

[CH3] Chandra, A.K., personal communication.

[C] Codd, E. F., A Relational Model for Large Shared Data Banks, Communications ACM 6,13 (June 1970)

[HU] Hopcroft, J.E., Ullman, J.D., Formal Languages and their Relation to Automata, Addison-Wesley, Reading, Mass. (1969)

[H] Hull, R. Relative information Capacity of Simple Relational Database Schemata. USC Technical Report 1984, To appear in SIAM J. of Computing.

[M] Maier, D., The Theory of Relational Databases, Computer Science Press (1983).

[MMS] Maier, D., A.O. Mendelzon, Y. Sagiv, Testing Implication of Data Dependencies, ACM Transactions on Database Systems (1979)

[MBH] Mylopoulos, J., P.A. Bernstein, H.K.T. Wong, A Language Facility for Designing Interactive Database-Intensive Applications, ACM Transactions on Database Systems (June 1980).

[P] Paredaens, J., On the Expressive Power of the Relational Algebra, Inform. Processing Letters (1978)

[SS] Sheard, T., D. Stemple, Automatic Verification of Database Transaction Safety, Technical Report, U. of Massachusetts, Amherst (1986)

[U] J.D. Ullman, Principles of Database Systems, 2nd edition, Computer Science Press (1982)

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique

the addition of a procedural element to strictly declarative programs. We believe that the languages defined in this paper provide some insight into how further increase in procedurality can increase the computational power of "declarative" languages. In particular, the language SdetTL appears to be useful in this context (recall that SdetTL is deterministic, does not use invented values, and is more powerful than stratified programs). We will further investigate this issue in future research.

Finally, note that, although defined for the relational model, our transaction languages can easily be extended to other models. For instance, for complex objects based on set and tuple constructors, on may consider the introduction of conditions of the form $x \in X$ in while statements. Analogous completeness results should be obtained for such extensions.

*Acknowledgement:* The authors would like to thank Ashok Chandra, Eric Simon and Moshe Vardi for useful discussions on this paper.

## REFERENCES

[AV1] Abiteboul, S., V. Vianu, Transactions in Relational Databases, Proc., Int'l Conf. on Very Large Data Bases (1984).

[AV2] Abiteboul, S., V. Vianu, Transactions and Integrity Constraints, Proc., ACM SIGACT/SIGMOD Symp. on Principles of Database Systems (1985).

[AV3] Abiteboul, S., V. Vianu, Deciding Properties of Transactional Schemas, Proc., ACM SIGACT/SIGMOD Symp. on Principles of Database Systems (1986).

[AV4] Abiteboul, S., V. Vianu, On Properties of Transactional Schemas, in preparation.

[AU] Aho, A.V., J.D. Ullman, Universality of Data Retrieval Languages, proc. 6th ACM Symp. on Principles of Prog. Languages, San Antonio, Texas (1979)

[B] Bancilhon, F., On the Completeness of Query Languages, proc. 7th Symp. on Mathematical Foundations of Computer Science, Zakopane, Poland (1978)

[BV] Beeri, C., M.Y. Vardi, Formal Systems for Tuple and Equality Generating Dependencies, Siam J. Computing, Vol. 13, No. 1 (1984)

[Br] Brodie, M.L., On Modelling Behavioral Semantics of Data, Proc., Int'l Conf. on Very